PROYECTO FIN DE CARRERA

GESTIÓN DE LAS RELACIONES CON EL CLIENTE MEDIANTE DISPOSITIVOS MÓVILES BLACKBERRY

Facultad de Informática
Universidad de Murcia
Junio 2010

AUTOR
Juan Antonio Jiménez Asencio

TUTOR
D. Francisco García Sánchez

ÍNDICE

ÍNDICE DE FIGURAS

1 INTRODUCCION

Es muy común cuando una persona se mueve en entornos empresariales escuchar a menudo el término **CRM**, pero, ¿qué es exactamente un CRM? CRM (**Customer Relationship Management**), que traducidas al castellano significan Gestión de las Relaciones con el Cliente, es decir, un CRM engloba los procesos relacionados con la gestión de los clientes dentro de una organización. El CRM no es una nueva filosofía de trabajo u organización, sino el resultado de unir las antiguas técnicas comerciales de los pequeños establecimientos con la tecnología de la información. El máximo objetivo del CRM es el de disponer, en cualquier momento, de toda la información sobre cualquier cliente, tanto para satisfacer las necesidades del cliente, como para obtener estudios de mercado que permitan unas mejores estrategias comerciales.

Una de las numerosas empresas que hace uso de CRM es **Nuevos Sistemas Tecnológicos (Neosistec)**, experta en desarrollo de aplicaciones para dispositivos móviles, especializándose en el desarrollo para dispositivos BlackBerry. Esta empresa cuenta con varios tipos de empleados, desde programadores y desarrolladores de software, expertos en sistemas y redes, hasta gerentes de recursos humanos, responsables de posventa y comerciales. Respecto a estos dos últimos (sobre todo los comerciales), una de las principales características a destacar es que están constantemente viajando para poder llevar los productos desarrollados por la empresa a los distintos nuevos clientes que puedan surgir, así como para atender las necesidades de los actuales clientes de la empresa, ya sea para informarles sobre nuevas actualizaciones de productos, como para reportar distintos fallos en los productos y/o servicios contratados. Para poder gestionar toda esa información hacen un uso intensivo de la herramienta SugarCRM, la cual es un CRM basado en la denominada LAMP, esto es, basado en las tecnologías Linux-Apache-MySql-Php, aunque también podría denominarse WAMP, Windows-Apache-MySql-Php, ya que también puede ser ejecutado en sistemas operativos Windows.

Para facilitar la movilidad de sus empleados, la empresa Neosistec ha solicitado el desarrollo de una aplicación para BlackBerry que fuera capaz de interactuar con SugarCRM para poder consultar y gestionar toda la información referente a cuentas, contactos, llamadas, etc., permitiendo de esta manera que todos sus comerciales, o cualquier otro empleado que haga uso de SugarCRM, puedan tener acceso a la información necesaria sobre sus clientes desde cualquier punto en el que se encuentren, sin necesidad de hacer uso de un ordenador para ello.

En esta memoria se detallan los pasos que se han llevado a cabo durante el desarrollo de la aplicación propuesta por la empresa. Para ello se comenzará explicando cuáles son los objetivos principales que se desean conseguir con la aplicación, así como las herramientas usadas para el desarrollo de la misma. Después se describe de forma detallada la implementación del proyecto, indicando el proceso de desarrollo de software usado, patrones de software

utilizado, así como ciertas particularidades que están presentes en el desarrollo de una aplicación para dispositivos móviles BlackBerry. Para terminar se comenta las conclusiones obtenidas tras el desarrollo del proyecto, las posibles mejoras que se podrán añadir al proyecto para futuras revisiones o mejoras de la aplicación y también se incluye un manual de usuario de la aplicación desarrollada.

2 OBJETIVOS Y METODOLOGIA

2.1 Objetivos del proyecto

La finalidad del proyecto descrito en esta memoria es la de construir una aplicación para dispositivos móviles BlackBerry que permita interactuar con la herramienta SugarCRM 5.5, en particular con la edición **Sugar Community Edition (SugarCE)**, que posee una licencia pública GPLv3. SugarCRM es una aplicación CRM Open Source, con una gran comunidad detrás de ella que está constantemente añadiendo nueva funcionalidad a la aplicación y corrigiendo los 'bugs' que aparecen. De esta manera, se pretende conseguir que desde cualquier punto en el que se encuentre un usuario con los privilegios para el acceso a la herramienta SugarCRM, pueda acceder a la información referente a cuentas, clientes, oportunidades, llamadas, reuniones, etc., desde su dispositivo móvil. Así, el usuario de la aplicación podrá estar en todo momento informado sobre los distintos cambios producidos en su entorno de trabajo de una manera rápida y sencilla. Además, el gerente de la empresa también se podría aprovechar de una aplicación con estas características para poder gestionar de una manera más eficiente el trabajo de sus empleados, pudiendo en cualquier momento comprobar si un empleado ha realizado una llamada planificada a un cliente, o si se ha llevado a cabo una reunión con algún cliente potencial y el resultado de la misma, asignar nuevas tareas a sus empleados, etc.

Comentar que, aunque podría accederse a toda esta información desde cualquier navegador Web, como por ejemplo el navegador de BlackBerry, SugarCE no está preparada para devolver la información en un formato que pueda ser mostrado dentro de las dimensiones que posee una pantalla de un dispositivo BlackBerry. Por ello, se deberá seleccionar la información más relevante para el usuario e intentar mostrarla de la manera más agradable y eficiente posible, dentro de las limitaciones de pantalla proporcionadas por el dispositivo. Además, también se deberá diseñar la aplicación siguiendo una estrategia que satisfaga los siguientes objetivos:

- ***Eficiencia en el uso disponible de conexión a la red***

  Hay que tener en cuenta que la mayoría de dispositivos que usarán esta aplicación tendrán una conexión GPRS, la cual tiene una velocidad de transferencia de hasta 144 kbps. Partiendo de esta limitación y con el objetivo de mejorar la experiencia del usuario en su interacción con el sistema, la aplicación deberá estar diseñada de manera que pueda almacenar la información obtenida desde SugarCE en el dispositivo, para que posteriores consultas sobre el mismo elemento no precisen una nueva llamada al servidor. Esto permitirá que el usuario obtenga una mayor fluidez en el uso de la aplicación, evitando los tiempos de esperas que

sufriría si intentará obtener esa información desde un navegador Web. Además, debemos pensar que ésta aplicación no va a ser la única que se ejecutará en el dispositivo, por lo que cuanto menos abusos se produzcan referentes al ancho de banda disponible, mejor será la sensación de correcto funcionamiento del dispositivo BlackBerry.

- ***Funcionamiento de la aplicación en modo offline***

  Otra de las características con que Neositec quiere dotar a la aplicación es el referente al modo de trabajo offline, es decir, que aunque en cualquier momento el dispositivo móvil se quede sin cobertura, el usuario pueda seguir teniendo acceso a la información, pudiendo consultarla, modificarla y/o borrarla, de manera que cuando el dispositivo vuelva a tener cobertura se envíen automáticamente los cambios realizados al servidor, siendo de esta manera accesible al resto de los usuarios. Lógicamente, cuando se habla de "tener acceso a la información" se entiende que es a los elementos que ya se encuentren descargados en el dispositivo.

- ***Actualización bi-direccional de la información***

  La aplicación deberá estar preparada no sólo para enviar los datos que se hayan modificado al servidor para que sean actualizados para el resto de los usuarios, sino que, además, se deberá diseñar la lógica necesaria para que cualquier modificación que se produzca en el servidor por otros usuarios sobre elementos que tengamos cacheados en el dispositivo sean automáticamente actualizados por la aplicación, sin necesidad de que el usuario realice acción alguna para ello. Además, esa actualización deberá realizarse de la manera más eficiente posible, no siendo concebible que, por ejemplo, la aplicación, para actualizar algunos pocos contactos que tenga descargados en el dispositivo, tenga que descargarse todos los contactos que haya en el servidor.

- ***Uso del servicio BlackBerry para acceso a la red***

  Como se ha comentado antes, la mayoría de dispositivos se conectan a la red a través de GPRS, la cual tiene la propiedad de poder tarificar por bytes transmitidos. Existen unas tarifas planas de datos proporcionadas por los proveedores de servicio que permiten tener acceso a Internet ilimitado pagando una cuota fija mensual. Neosistec está interesado en que la aplicación diseñada solo pueda acceder a Internet a través de ese servicio de datos de tarifa plana, evitando de esta manera que se produzcan facturas desorbitadas por el uso de la misma. Para ello se diseñará la aplicación de manera que las conexiones con el servidor solo se hagan a través de este servicio, impidiendo que la aplicación pueda salir por otros **APN** (Acces Point Name, nombre de punto de acceso) los cuales producirían la tarificación por bytes transmitidos. Por otro lado, Neosistec

también ha indicado que sería interesante que la aplicación también pudiera, de manera opcional, tener acceso a la red a través de WIFI.

- ***Minimizar el consumo de batería del dispositivo***

  Este punto está directamente relacionado con los tres primeros, ya que cuanto más uso haga la aplicación de la radio para enviar y recibir la información del servidor, mayor consumo de batería tendrá. Este aspecto es crítico, ya que como se ha comentado en la introducción, esta aplicación está pensada para usarse en "movilidad", por lo que no es factible que debido a un uso excesivo de la radio por parte de nuestra aplicación, el consumo de batería crezca exponencialmente. Para ello, se diseñaran las políticas necesarias para intentar minimizar las llamadas realizadas desde nuestra aplicación al servidor, reduciendo de esta manera el consumo de batería de los dispositivos BlackBerry.

- ***Controlar el uso de memoria consumida***

  Como ya se ha mencionado, la aplicación tendrá la posibilidad de almacenar en la memoria persistente del dispositivo la información descargada desde el servidor. Luego, hay que tener en cuenta que otra característica importante de todos los dispositivos móviles es que el tamaño de la memoria disponible es muy limitado (aunque bien es cierto que los últimos modelos han mejorado considerablemente este apartado, llegando a tener una capacidad de 2 GB). Las BlackBerry sobre las que inicialmente se ha ejecutado la aplicación desarrollada tienen una memoria persistente cuyo tamaño varía entre los 16 y 128 MB de capacidad, según el modelo de dispositivo, por lo que se ha tenido que diseñar la aplicación de manera que sea capaz de liberar información que ya no sea relevante para el usuario. Del mismo modo, habrá que considerar políticas de liberación de memoria en caso de que sea obligatorio realizarla porque la memoria esté llegando al límite de su capacidad.

Estos son los principales objetivos que la aplicación debe alcanzar. Para ello, se hace uso de las distintas herramientas disponibles para el desarrollo de aplicaciones BlackBerry, así como de otras de carácter general, que nos permiten desarrollar una aplicación capaz de alcanzar todos los objetivos propuestos.

A continuación, se describe la metodología usada para el diseño de la aplicación, y se comentan las distintas tecnologías utilizadas para su desarrollo. También se mencionan algunas propiedades específicas sobre los dispositivos móviles BlackBerry que deben tenerse en cuenta para poder realizar el desarrollo de una aplicación para este tipo de dispositivos.

2.2 Metodología

Para un correcto diseño y desarrollo de software es necesario aplicar un proceso de desarrollo del software, el cual describe un enfoque para la construcción, desarrollo y, posiblemente, mantenimiento del software. Existen diferentes procesos de desarrollo de software, pero para este proyecto se ha elegido el *Proceso Unificado (UP) (Larman, 2003)*, el cual combina las prácticas comúnmente aceptadas como "buenas prácticas", tales como ciclo de vida iterativo y desarrollo dirigido por el riesgo, en una descripción consistente y bien documentada.

De las buenas prácticas que fomenta el UP, una destaca sobre las demás: el desarrollo iterativo. En este enfoque, el desarrollo se realiza en una serie de mini-proyectos cortos, de duración fija que se denomina iteraciones. El resultado de cada una de estas iteraciones es un sistema que puede ser probado, integrado y ejecutado. Cada iteración incluye sus propias actividades de análisis de requisitos, diseño, implementación y pruebas. El ciclo de vida iterativo se basa en la ampliación y refinamiento sucesivos del sistema mediante múltiples iteraciones, con retroalimentación cíclica y adaptación como elementos principales que dirigen para converger hacia un sistema adecuado. El sistema crece incrementalmente a lo largo del tiempo, iteración tras iteración, y por ello este enfoque también se conoce como desarrollo iterativo e incremental.

Esto implica un aspecto crucial en el desarrollo de software: la aceptación de los cambios. Lo que se intenta evitar es especificar y congelar todos los requisitos del sistema, aceptando que estos cambiarán conforme el personal involucrado clarifica su visión del mismo o cambia el mercado.

El UP organiza el trabajo y las iteraciones en cuatro fases fundamentales:

1) **Inicio:** visión aproximada, análisis del negocio, alcance, estimaciones imprecisas.

2) **Elaboración:** visión refinada, implementación iterativa del núcleo central de la arquitectura, resolución de los riesgos altos, identificación de más requisitos y alcance, estimaciones mas realistas.

3) **Construcción:** implementación iterativa del resto de requisitos de menor riesgo y elementos más fáciles, preparación para el despliegue.

4) **Transición:** pruebas beta, despliegue.

Esto no corresponde con el antiguo ciclo de vida en cascada o secuencial, en el que primero se definían todos los requisitos y, después, se realizaba todo, o la mayoría, del diseño. La fase de inicio no es una fase de requisitos, sino una especie de fase de viabilidad, donde se lleva a cabo solo el estudio suficiente para decidir si continuar o no. De igual modo, la fase de Elaboración no es la fase de

requisitos o de diseño, sino que es una fase donde se implementa, de manera iterativa, la arquitectura que constituye el núcleo central y se mitigan las cuestiones de alto riesgo.

2.3 Tecnologías utilizadas

2.3.1 Sugar Community Edition

Tal y como ya se ha comentado anteriormente, la aplicación BlackBerry deberá interactuar con la aplicación SugarCRM (sugarcrm, 2010). La principal característica de esta herramienta es que es Open Source, por lo que podremos descargarnos el código fuente del mismo y, si fuera necesario, modificarlo para adaptarlo a nuestras necesidades, aunque esto se evitará en lo posible, ya que el objetivo del proyecto es que la aplicación BlackBerry funcione con cualquier SugarCRM. Por ello, lo primero que deberíamos conocer es cómo está formado SugarCRM, y que características tiene que sean interesantes para un desarrollador.

SugarCRM está formado por módulos, cada uno de los cuales representa una función específica de un CRM, por lo que podemos encontrarnos módulos que representan Cuentas, Contactos, Oportunidades, Reuniones, Llamadas, etc. Por ejemplo, el módulo de *Cuentas* permite crear y gestionar las cuentas de los clientes, y el módulo de *Llamadas* permite gestionar las llamadas relacionadas con cuentas, oportunidades, contactos, etc. Estos módulos están diseñados para ayudar al usuario en la gestión de las distintas etapas del ciclo de vida del cliente, iniciándose con la generación de un cliente potencial y su posterior conversión a cliente, hasta el reporte y resolución de las distintas incidencias que puedan producirse con un cliente. Como las etapas de ese ciclo están relacionadas, cada módulo mostrará las relaciones que tiene con los demás. Por ejemplo, cuando el usuario esté visualizando los detalles de una cuenta, ésta le mostrará los contactos, llamadas, reuniones, oportunidades, etc., que estén relacionados con dicha cuenta.

Para visualizar toda esta información, el usuario puede utilizar cualquier navegador Web desde el que podrá acceder a SugarCRM y, navegando por las distintas ventanas que va mostrando la aplicación, acceder a los distintos módulos que componen la aplicación. Como ejemplo, en la Figura 1 se muestra la página principal que presenta SugarCRM una vez autenticado en el sistema.

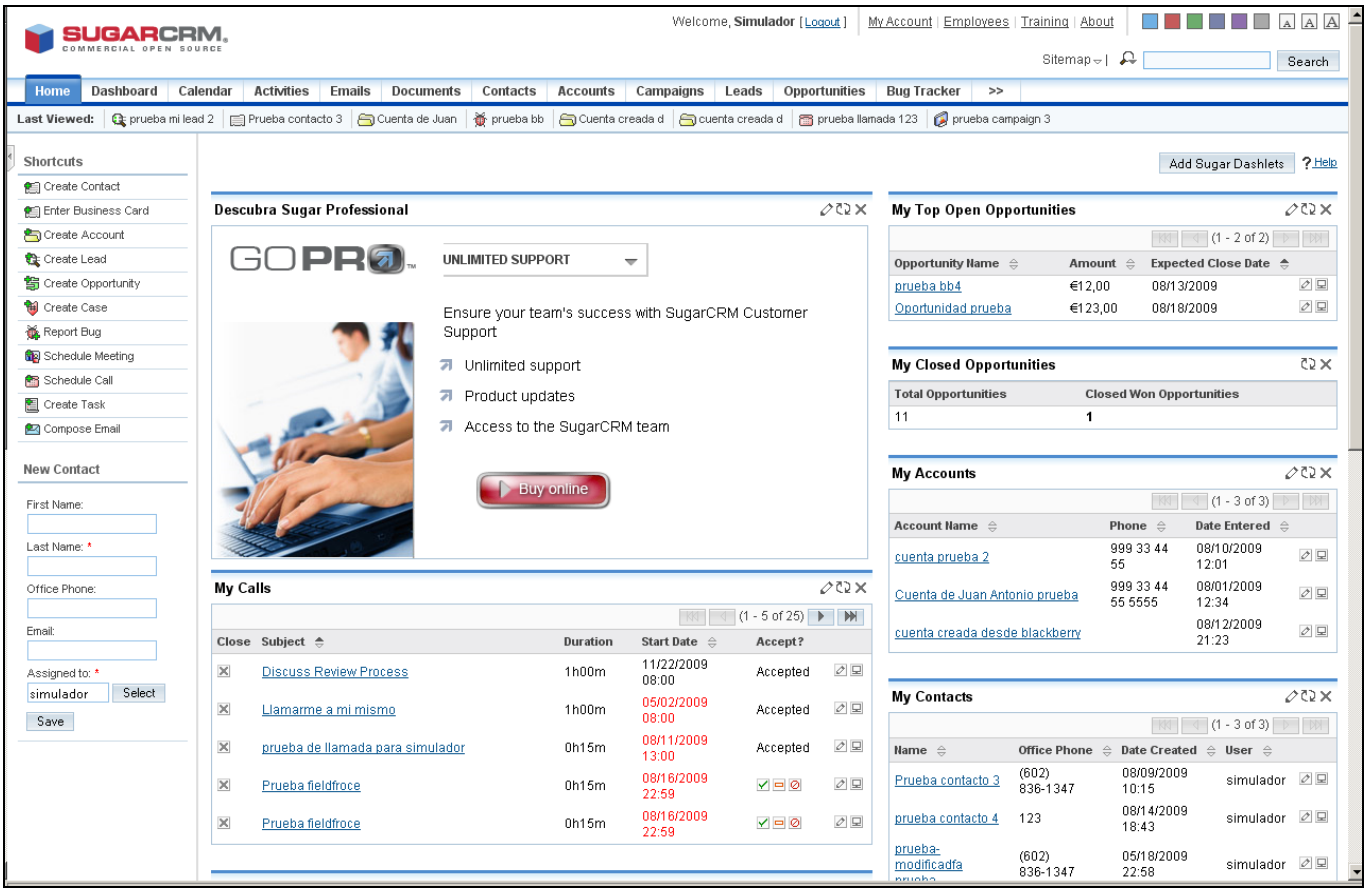

Figura 1. Pantalla principal de SugarCRM

En la citada imagen podemos ver las distintas pestañas que representan los módulos de la aplicación. Si pulsamos en cada una de ellas nos llevara a la pantalla de listado del módulo, desde la cual podremos gestionar los elementos de ese módulo.

Una vez que se ha introducido esta breve explicación de la parte de usuario, se pasará a la parte más interesante, la correspondiente al desarrollador. Como ya se ha comentado, SugarCRM es una aplicación Open Source, por lo que podemos tener acceso al código para poder añadirle nuevas funcionalidades, así como para arreglar los "bugs" que puedan aparecer en la aplicación. SugarCRM está disponible en tres ediciones: la Community Edition, la Professional Edition y la Enterprise Edition. De estas tres ediciones, se ha trabajado con la primera edición, que está disponible para su descarga y uso de forma gratuita. Todas las ediciones han sido desarrolladas por el mismo equipo de desarrollo, por lo que la base del código fuente de la aplicación es la misma para las tres ediciones, diferenciándose las versiones de pago de la libre en ciertos módulos específicos que añaden cierta funcionalidad avanzada a la aplicación para la gestión del negocio.

Desde los inicios de SugarCRM, el equipo de desarrollo diseñó la aplicación teniendo en mente que el código fuente sería examinado y modificado por otros desarrolladores. Por ello, la aplicación ha sido desarrollada de manera que

cualquier modificación sobre cualquier módulo de la misma pueda ser realizada de un modo seguro, sin que ello afecte al funcionamiento de otros módulos, o incluso a la del propio módulo modificado (lógicamente, se refiere a las partes del módulo que no sean modificadas). SugarCRM está basado en un framework modular que posee ciertos puntos de entrada seguros a la aplicación, como por ejemplo *"index.php"*. La estructura de directorio en la que reside el código de la aplicación es la siguiente:

- **Cache:** En este directorio se almacenan ficheros que permiten minimizar el acceso a la base de datos, así como plantillas de las interfaces de usuario creadas a partir de los ficheros de metadatos. Además, se almacenan otra clase de archivos como Notas o Documentos creados por los usuarios y añadidos a la aplicación, por lo que no todos los ficheros de este directorio deben borrarse.

- **Custom:** Directorio usado para la modificación de la aplicación de forma segura. Aquí crearemos nuestras definiciones de campos propias, diseño de nuevas interfaces, etc.

- **Data:** Los principales ficheros de la aplicación se almacenan aquí. Destacar el fichero con la clase base SugarBean, la cual controla la lógica por defecto de todos los objetos de negocio incluidos en la aplicación.

- **Include:** Varias funciones útiles se localizan aquí, así como otras librerías que ayudan a Sugar a realizar parte del trabajo. Destacar de aquí el fichero utils.php, que contiene varias de las funciones de ayuda usadas por Sugar.

- **Metadata:** Aquí se encuentran los ficheros que contienen la información necesaria para trabajar con las relaciones de tipo muchos-a-muchos existentes entre los objetos de negocio.

- **Modules:** Aquí se encuentran todos los módulos del sistema. Los módulos propios que han sido instalados usando la herramienta "*Module Loader*" también se localizan aquí.

Todos los módulos, tanto los que ya están incluidos en la aplicación, como aquellos que sean construidos por nosotros mismos, deben estar localizados en la carpeta "*<directorio-raíz>/modules/"*.

El acceso a SugarCRM se realiza a través de una serie de puntos de entrada. El principal punto, que será el usado por la mayoría de usuarios, corresponde al *"index.php"*, que se encuentra localizado en el directorio raíz de Sugar. Los principales parámetros para la mayoría de llamadas que se realicen a Sugar son los siguientes:

- **Module:** El módulo al que se desea tener acceso.

- **Action:** La acción a realizar por la aplicación dentro de ese módulo.

- **Record:** El identificador del registro al que se quiere acceder.

  Así, un ejemplo de llamada a la aplicación sería el siguiente:

  http://sugardev.neosistec.com/index.php?module=Accounts&
  action=DetailView&record=147c09e9-1897-7615-f6e3-4a8302cd9ab8

Otro punto de entrada corresponde al fichero *"soap.php"*, del cual hablaremos más adelante.

A continuación se describen brevemente cómo están diseñados los módulos que componen la aplicación. Un módulo de SugarCRM está formado por los siguientes elementos:

- **Vardefs.php:** Un fichero que contiene información referente a la tabla usada en la base de datos, campos, tipos de datos y relaciones.

- **SugarBean:** SugarBean es la clase base para todos los objetos de negocio de Sugar. Esta clase implementa la funcionalidad para crear, recibir, actualizar, y borrar objetos en Sugar. Cada módulo debe implementar esta clase base para añadir los campos y funcionalidad propia de este módulo.

- **Ficheros de metadatos:** Estos ficheros contienen la información necesaria para definir el contenido y diseño de las ventanas de Sugar:

  - **ListView:** Lista los registros existentes en el módulo.

  - **DetailView:** Muestra los detalles de un registro.

  - **EditView:** Permite al usuario editar un registro.

  - **SubPanels:** Muestran las relaciones de un registro del modulo con los registros de otros módulos.

  - **Popups:** Muestran la lista de registros que pueden ser enlazados a otro registro.

Para terminar, comentar que SugarCRM fue diseñado haciendo uso del patrón Modelo-Vista-Controlador. De esta forma se consigue una separación entre la lógica de negocio y la lógica de presentación. En SugarCRM, se aplica el patrón de la siguiente forma:

- **Modelo:** El modelo es representado por el SugarBean y todas sus subclases.

- **Controlador:** SugarCRM introduce un concepto de controlador en cascada, lo cual permite incrementar la granularidad sobre las personalizaciones de los desarrolladores, así como proveer capacidades adicionales para que se

realicen de modo seguro. El controlador principal, llamado SugarController, direcciona las acciones básicas de las vistas de Edit y DetailsView hacia el modelo. Cada módulo puede extender el controlador principal para añadir la funcionalidad necesaria.

- **Vista:** Las vistas son las responsables de mostrar la información en los navegadores. Al igual que en el controlador existe una clase por defecto llamada SugarView que implementa una parte de la lógica básica para las vistas, encargándose por ejemplo de las cabeceras y pie de página.

Es posible encontrar más información relativa a SugarCRM en la página de SugarForce http://www.sugarforge.org/.

2.3.2 Servicios Web

Para el desarrollo del proyecto se han hecho uso de los servicios Web (w3, 2010) proporcionados por SugarCRM que están formados por una serie de funciones que permiten al dispositivo BlackBerry interactuar con el servidor SugarCRM, permitiéndole realizar login/logout, descarga de registros de un módulo, creación/modificación/eliminación de un registro, etc. Por lo tanto, lo primero que se debería hacer es saber que es un servicio Web.

Existen múltiples definiciones sobre lo que son los Servicios Web, lo que muestra su complejidad a la hora de dar una adecuada definición que englobe todo lo que son e implican. Una posible sería hablar de ellos como un conjunto de aplicaciones o de tecnologías con capacidad para interoperar en la Web. Estas aplicaciones o tecnologías intercambian datos entre sí con el objetivo de ofrecer unos servicios. Los proveedores ofrecen sus servicios como procedimientos remotos y los usuarios solicitan un servicio llamando a estos procedimientos a través de la Web. Luego las principales funciones que desempeña un servicio Web son, por un lado, permitir la reutilización de componentes ya generados en nuestra aplicación, y por otro, conectar software existente en distintas plataformas, resolviendo el problema que normalmente solían tener dos aplicaciones que habían sido desarrolladas en plataformas distintas, o usando lenguajes de programación distintos, cuando intentaban intercambian datos entre ellas.

La tecnología de servicios Web está basada en tres estándares fundamentales, a saber, SOAP, WSDL y UDDI. A continuación, se describe el propósito de cada uno de estos estándares.

2.3.2.1 SOAP

SOAP es un simple protocolo basado en XML que permite a las aplicaciones el intercambio de información sobre HTTP, es decir, SOAP es un protocolo que permite el acceso a los servicios Web.

Un mensaje SOAP es un documento XML que contiene los siguientes elementos:

- El elemento *"Envelope"*, que identifica el documento XML como un mensaje SOAP.
- El elemento *"Header",* que contiene la información de cabecera.
- El elemento *"Body",* que contiene la información de la llamada y de respuesta.
- El elemento "Fault", que contiene información sobre los errores producidos.

Así, un ejemplo de de la estructura de un mensaje SOAP sería el siguiente:

```
<?xml version="1.0" ?>

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

   <soap:Header>...</soap:Header>

   <soap:Body>
        ...
      <soap:Fault>...</soap:Fault>
  </soap:Body>

</soap:Envelope>
```

Las principales características de los elementos que forman parte de un mensaje SOAP son las siguientes:

- **Elemento Envelope**

  "Envelope" es el elemento raíz de un mensaje SOAP y define el documento XML como un mensaje SOAP. Este elemento consta de dos atributos que son el *"namespace"* y el *"encodeStyle"*. El namespaces define el elemento como un SOAP Envelope, y siempre debe tener el valor *"http://www.w3.org/2001/12/soap-envelope"*, ya que si no la aplicación que esté esperando para recibir mensajes SOAP generaría un

error y descartaría el mensaje. Por otro lado, el atributo encodingStyle es usado para definir los tipos de datos del documento. Este atributo, al contrario que el anterior, no es exclusivo del elemento Envelope, ya que puede aparecer en cualquier elemento del mensaje SOAP, de modo que los tipos de datos que defina serán aplicados para ese elemento y sus hijos. Un ejemplo de este elemento sería el siguiente:

```
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-
   envelope"
   soap:encodingStyle="http://www.w3.org/2001/12/soap
   -encoding">
   ……
   El resto del mensaje
   ……
 </soap:Envelope>
```

- **Elemento Header**

"*Header*" contiene información específica para la aplicación sobre el mensaje SOAP. Este elemento es opcional y, si aparece, debe ser el primer elemento hijo del elemento Envelope. Como todo elemento xml, se pueden definir atributos en la cabecera. Los atributos especificados definen cómo debe procesar un receptor el mensaje SOAP. En la cabecera SOAP aparecen tres atributos en su espacio de nombres por defecto (http://www.w3.org/2001/12/soap-envelope) llamados m*ustUnderstand*, *actor* y *encodingStyle*.

1. **mustUnderstand:** Este atributo indica si un elemento de la cabecera debe ser procesado obligatoriamente por el receptor o si, por el contrario, es opcional. Si un elemento de la cabecera tiene establecido este atributo a "1" y el receptor no lo reconoce, entonces el receptor fallará al procesar la cabecera.

2. **actor:** Este atributo indica qué receptor debe procesar este elemento de la cabecera. Hay que tener en cuenta que cuando se envía un mensaje SOAP entre un emisor y un receptor, este mensaje puede pasar por varios puntos intermedios antes de llegar a su destinatario. Este atributo indica quién de todos ellos debe procesar ese elemento.

3. **encodingStyle:** Realiza la misma función que para el elemento Envelope.

- **Elemento Body**

Este elemento contiene el mensaje que será procesado por el receptor, si se trata de una petición, o por el emisor, si se tratara de la respuesta. Los elementos hijos que se incluyan son específicos de cada aplicación, es decir, no forman parte del espacio de nombres de SOAP. Un ejemplo de mensaje de petición y respuesta sería el siguiente:

```
<?xml version="1.0" ?>
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   soap:encodingStyle="http://www.w3.org/2001/12/soap-
   encoding">
    <soap:Body>
      <m:Test
      xmlns:m="http://www.ejemlpo.es/test">
         <m:Frase>Test de SOAP</m:Frase>
      </m:Test>
    </soap:Body>
 </soap:Envelope>
```

```
<?xml version="1.0" ?>
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   soap:encodingStyle="http://www.w3.org/2001/12/soap-
      encoding">
  <soap:Body>
      <m:TestResponse
      xmlns:m="http://www.ejemplo.es/test">
         <m:FraseDevuelta>
              Su frase: Test de SOAP
         </m:Item>
       </m:Test>
   </soap:Body>
 </soap:Envelope>
```

En el ejemplo anterior, el cliente realiza una llamada a la función *Test*, pasándole un parámetro a la misma incluido en el elemento <Frase>. El receptor responde a ese mensaje enviando otro mensaje de respuesta TestResponse, el cual incluye la respuesta del servidor en el elemento <FraseDevuelta>.

- **Elemento Fault**

Este elemento contiene los errores que se pueden producir al procesar un mensaje. Este elemento es opcional y, si está presente en el mensaje, debe aparecer como elemento hijo del elemento Body, y

sólo puede aparecer una vez. El elemento Fault contiene los siguientes subelementos:

1) **faultcode:** Un código que identifica el fallo.

2) **faultstring:** Contiene una breve explicación sobre el error.

3) **faultactor:** Información sobre quién causó este error.

4) **detail:** Contiene información específica de la aplicación sobre el error ocurrido.

Los posibles valores que pueden ser asignados al elemento <*faultcode*> son los siguientes:

- **VersionMisMatch:** Indica que el atributo "namespace" es incorrecto para el elemento <Envelopment>.
- **MustUnderstand:** Indica que uno de los hijos obligatorios de la cabecera no ha podido ser procesado.
- **Client:** Indica que el mensaje está mal formado o contiene información incorrecta.
- **Server:** Indica que se produjo un problema en el servidor que impidió procesar el mensaje.

Un mensaje SOAP puede viajar a través de la red encapsulado en HTTP, SMTP, etc. Aunque bien es cierto que lo más normal es que vaya encapsulado en HTTP. Cuando ocurre esto, el mensaje HTTP puede crearse usando el método GET o el POST, pero si se usa este último debe especificarse al menos las cabeceras *Content-Type* y *Content-Length*. La primera cabecera define el tipo MIME del mensaje y, opcionalmente, el tipo de codificación de caracteres usado por el cuerpo XML del mensaje para la petición o la respuesta. La segunda cabecera indica el numero de bytes que ocupa el cuerpo del mensaje o la respuesta.

2.3.2.2 WSDL

WSDL es un documento escrito en XML que describe a un servicio Web, especificando la localización del servicio y las operaciones que el servicio expone. Para poder describir un servicio Web, el documento WSDL hace uso principalmente de cuatro elementos:

- **types**

  Este elemento define los tipos de datos que son usados por el servicio Web. Para conseguir la mayor interoperabilidad posible entre las distintas plataformas, WSDL usa la sintaxis del XML Schema para definir los tipos de datos.

- **message**

  Este elemento define los datos que forman parte de una operación. Cada mensaje puede contener una o mas "partes". Las partes pueden compararse a los clásicos parámetros de una función de cualquier lenguaje de programación tradicional.

- **portType**

  Este elemento es el más importante de todos. Se encarga de definir las operaciones que pueden ser realizadas, así como los mensajes que están involucrados en cada operación. Los tipos de operaciones definidos son los siguientes:

  - **One-Way (Una direccion):** La operación puede recibir mensajes pero no devolverá una respuesta.

  - **Request-Response(Petición-Respuesta):** La operación puede recibir una petición y devolverá una respuesta.

  - **Solicit-Response (Solicitud-Respuesta):** La operación puede enviar una petición y esperará una respuesta.

  - **Notification (Notificación):** La operación puede enviar un mensaje pero no esperará a una respuesta.

  De todas ellas, las operaciones de tipo petición-respuesta suelen ser las más frecuentes.

- **binding**

  Este elemento define el formato del mensaje y los detalles del protocolo usado para un servicio Web. Este elemento tiene dos atributos: "*name*" y "*type*". El atributo "name" provee un nombre único entre todos los "bindings" existentes en el documento WSDL, y el atributo "type" indica cuál es el puerto al que está enlazado. Además de estos atributos, contiene dos elementos hijos que son:

  - **soap:binding:** Elemento compuesto por dos atributos: "*style*" y "*transport*". El primer atributo indica el estilo del mensaje, el cual

puede tomar los valores de rpc o document. El segundo atributo indica cuál es el protocolo en el que se encapsulará el mensaje SOAP, como por ejemplo HTTP, SMTP, etc.

- **operation:** Elemento que define cada una de las operaciones que se definen en un puerto. Para cada operación se le definirá su correspondiente acción SOAP, y se indicará el tipo de codificación usada para los mensajes de entrada y salida.

En el ANEXO II – EJEMPLO FICHERO WSDL se muestra un ejemplo de configuración de fichero WSDL.

2.3.2.3 UDDI

UDDI es un registro público diseñado para almacenar de forma estructurada información sobre empresas y los servicios que éstas ofrecen. A través de UDDI, se puede publicar y descubrir información de una empresa y de sus servicios. Se pueden utilizar sistemas estándares para clasificar estos datos y poder encontrarlos posteriormente en función de la categorización. Lo más importante es que UDDI contiene información sobre las interfaces técnicas de los servicios de una empresa. A través de un conjunto de llamadas basadas en SOAP, se puede interactuar con UDDI tanto en tiempo de diseño como de ejecución para descubrir datos técnicos de los servicios que permitan invocarlos y utilizarlos. De este modo, UDDI sirve como infraestructura para una colección de software basado en servicios Web.

El registro de un negocio en UDDI tiene tres partes:

- **Páginas blancas**: Dirección, contacto y otros identificadores conocidos.
- **Páginas amarillas:** Categorización industrial basada en taxonomías.
- **Páginas verdes:** Información técnica sobre los servicios que aportan las propias empresas.

La publicación en UDDI es un proceso relativamente sencillo. El primer paso consiste en determinar información básica sobre cómo definir la empresa y los servicios en UDDI. El siguiente paso, una vez determinada esta información, consiste en llevar a cabo el registro, ya sea mediante programación o a través de una interfaz de usuario basada en el Web. Por último, se debe probar la entrada para asegurar que se registró correctamente y que aparece tal y como se esperaba en diferentes tipos de búsquedas y herramientas.

2.3.2.4 Servicios Web de SugarCRM

SugarCRM provee un servicio Web construido usando la librería NuSOAP y disponible para las tres ediciones de Sugar. NuSOAP es un kit de herramientas (ToolKit) para desarrollar servicios Web bajo el lenguaje PHP. Está compuesto por una serie de clases que facilitan el desarrollo de servicios Web. Provee soporte tanto para el desarrollo del lado del cliente (aquellos que consumen los servicios Web), como del lado del servidor (aquellos que los proveen). NuSOAP está basado en SOAP 1.1, WSDL 1.1 y HTTP 1.0/1.1.

Para poder ver la API de SugarSOAP tan solo se debe escribir en un navegador la dirección http://www.ejemplo.es/sugar/soap.php. El fichero WSDL se encuentra localizado en http://www.ejemplo.es/sugar/soap.php?wsdl.

2.3.3 KSOAP y KXML

Se ha comentado que las comunicaciones entre la aplicación BlackBerry y el servidor SugarCRM se realizarán a través de los Servicios Web proporcionados por SugarCRM, por lo que la primera idea en la que se puede pensar es en implementar el código que haga uso de esos servicios utilizando los paquetes Xerces(para traducir XML) y Axis(para la creación de mensajes SOAP), pero aquí surge un pequeño problema, y es que estos paquetes fueron diseñados en su momento para clientes "desktop" y servidores, por lo que al intentar usar esto paquetes en dispositivos móviles que usan tecnología J2ME nos encontramos con dos problemas:

1. ***El tamaño de los paquetes es excesivo.*** Hay que pensar en que la memoria flash de los dispositivos móviles es muy escasa, por lo que no es factible hacer uso en nuestras aplicaciones de paquetes que pueden llegar a ocupar más de 1 MB para realizar la función de traducir texto XML, ya que el tamaño de la aplicación final podría ser demasiado grande para poder ejecutarse en el dispositivo.

2. ***No existen todas las clases usadas por esos paquetes en J2ME.*** Debido a las limitaciones propias en memoria y capacidad de procesamiento de los dispositivos móviles, las máquinas virtuales de los mismos son versiones reducidas de las originales máquinas virtuales (J2SE, J2EE). Debido a esto es posible que clases que puedan usar los paquetes Xerces o Axis directamente no existan en la máquina virtual, por lo que no podrían ejecutarse.

Por suerte, existen dos librerías diseñadas por Enhydra.org que se encargan de realizar tanto el trabajo de traducción de XML como de la creación de mensajes SOAP. Estas librerías son KXML y KSOAP (ksoap2, 2010). Estas librerías han sido diseñadas desde un principio para que puedan ser ejecutadas en dispositivos móviles, por lo que usan clases que están incluidas en J2ME. El tamaño de ambas librerías ocupa unos 50 KB, tamaño razonable para poder ser usada en aplicaciones para dispositivos limitados en memoria.

2.3.4 Ciclo de vida del desarrollo de aplicaciones BlackBerry

El desarrollo de aplicaciones para los dispositivos Blackberry tiene una metodología parecida en las etapas iniciales a la creación de aplicaciones de servidor o desktop, en el sentido de que pueden aplicarse los mismos procesos de desarrollo de software, aunque siempre teniendo en cuenta las limitaciones técnicas que conllevan los dispositivos móviles (Kao & Sarigumba, 2009). En lo que sí se diferencia sería en las fases correspondientes a la preverificación del *jar*, creación del archivo *jad* y *axl*, creación del archivo *cod* y firma del archivo *cod* generado. Por lo tanto las etapas serían las siguientes:

I. ***Captación de requisitos, Análisis, Diseño e Implementación de la aplicación***. Esta etapa es parecida a cualquier desarrollo de software realizada para cualquier plataforma, donde se puede (y se debe) utilizar procesos de desarrollo de software como el Proceso Unificado, o el Extreme Programming, u otros más pesados como Métrica.

II. ***Creación del archivo jar del código implementado.*** Compilación de la aplicación y empaquetado de la misma en un fichero jar usando J2ME, es decir, con una versión especifica de CLDC (corresponde a una configuración específica de J2ME dedicada a dispositivos con estrictas limitaciones de memoria, capacidad de cálculo, consumo y conectividad de red) y MIDP (corresponde a un perfil que extiende la configuración CLDC, suministrando una interfaz de usuario, métodos de entrada y mecanismos de persistencia, así como un entorno de desarrollo completo para un conjunto específico de dispositivos, soportando sus características concretas). En nuestro caso se usará una CLDC específica proporcionada por RIM que recibe el nombre de CLDC Application.

III. ***Preverificación del archivo jar generado.*** El paso que sigue a la compilación es preverificar las clases que se han generado. Sin esta preverificación no será posible la ejecución de la aplicación en el dispositivo. La preverificación nos asegura que no existe ningún tipo de código malintencionado que pueda dañar o crear un comportamiento extraño en nuestro dispositivo o en la máquina virtual, ya que debido a que los dispositivos J2ME no tienen demasiada capacidad de proceso, es necesario que la máquina virtual sobre la que se ejecutan las aplicaciones sea lo más eficiente posible, por lo que se han eliminado muchas de las comprobaciones que realizan las máquinas virtuales habitualmente.

IV. ***Creación del archivo cod.*** Una vez que el *jar* esta creado se crea el archivo *cod*, el cual es parecido al archivo *jar* (en el sentido de que contiene todas las clases y recursos que utiliza la aplicación) pero con un formato específico para BlackBerry. Estos archivos son los que más tarde se cargarán en el dispositivo y se ejecutaran en la máquina virtual del mismo.

V. ***Creación de los archivos jad y alx.*** Estos archivos son utilizados para desplegar los archivos cod que forman parte de la aplicación en el dispositivo. El archivo *jad*, usado para ser instalado a través de OTA (Over-the-air), contiene información necesaria para la instalación de los archivos contenidos en el archivo *cod*. El archivo *cod* puede contener dentro de él varios *cod*. Cuando ocurre esto, hablamos de una suite, y en el fichero *jad* se indicaran cada uno de los *cod* que forman parte de la suite. Los archivos alx también contiene información sobre el *cod* creado y es usado por la aplicación BlackBerry Desktop Manager para instalar la aplicación desde el PC a la BlackBerry mediante el puerto USB.

VI. ***Firma del archivo cod.*** Cuando se desarrollan aplicaciones para BlackBerry y se hace uso de las librerías propias desarrolladas por RIM, las cuales nos permiten entre otras cosas hacer uso de componentes UI ya desarrollados, librerías para interactuar con otras aplicaciones ya instaladas en la BlackBerry (teléfono, Mensajes, Calendario, etc.), acceder a la memoria persistente, etc., es necesario que RIM firme el archivo *cod* generado para que la aplicación pueda ejecutarse en el dispositivo. Si no se firmara la aplicación, al intentar ejecutarla en el dispositivo se lanzaría una excepción indicando que la aplicación hace uso de clases que necesitan ser firmadas para ejecutarse.

VII. ***Despliegue de la aplicación.*** Una vez desarrollada la aplicación y generado y firmado el archivo *cod*, lo último que queda es desplegarla en el dispositivo para su ejecución. A día de hoy existen 5 maneras para desplegar una aplicación en la BlackBerry, que son:

- *Usando la utilidad javaloader:* Un usuario puede conectar la Blackberry a su PC a través del USB y ejecutar desde la consola el comando *javaloader* para así cargar la aplicación creada en el dispositivo. Si bien este método solo es usado por los desarrolladores para cargar y testear rápidamente una aplicación creada.

- *Desktop:* La aplicación puede ser descargada por el usuario a su PC y, a través del BlackBerry Desktop Manager, cargarla vía USB en el dispositivo. Comentar que para el caso anterior también es necesario instalar el BlackBerry Desktop Manager ya que contiene los drivers para conectar el dispositivo al PC vía USB. Como ya se ha comentado antes aquí entra en juego el archivo *alx* creado previamente para realizar la instalación en el dispositivo.

- *OTA (Over-the-air):* La aplicación es descargada usando el BlackBerry Browser (que ya se encuentra instalado en la BlackBerry) desde un sitio web que se encuentre en Internet o una intranet. Como ya se ha comentado antes, aquí entra en juego el archivo *jad* creado previamente para realizar la descarga de la aplicación y su posterior instalación en el dispositivo.

- *BES push:* En entornos empresariales donde se haya instalado un BlackBerry Enterprise Server (BES), el administrador del mismo puede subir las aplicaciones creadas al BES y este automáticamente instalará la aplicación en los dispositivos que tenga asignados, sin que el usuario de dispositivo tenga que realizar acción ninguna.

- *BlackBerry App World:* Es una aplicación desarrollada por RIM que cualquier usuario puede instalar en su dispositivo (por medio de

alguno de los métodos anteriores) y que le permite visualizar e instalar las distintas aplicaciones que se encuentran en un servidor administrado por RIM, en el cual cualquier desarrollador, previamente registrado, puede colgar sus aplicaciones.

2.3.5 BlackBerry JDE y BlackBerry JDE Plug-in para Eclipse

Existen dos entornos de desarrollo para BlackBerry producidos por RIM: el BlackBerry Java Development Environment (JDE) y el BlackBerry JDE Plug-in para Eclipse. Ambos son funcionales y han sido usados por desarrolladores para producir aplicaciones profesionales. Si bien el JDE está algo más maduro, el Plug-in para Eclipse ha avanzado mucho, teniendo ya una versión release de la misma, por lo que a la hora de seleccionar un entorno u otro premiará más la preferencia personal de cada desarrollador que la potencia de una u otra herramienta. Para el desarrollo de este proyecto se ha hecho uso del Plug-in para Eclispe, ya que tenía experiencia con esta herramienta, y los componentes de las distintas versiones del SO (Sistema Operativo) para BlackBerry funcionan perfectamente, aunque eso sí, solo están disponibles desde la versión del SO 4.2 hacia arriba, por lo que si surgiera la necesidad de desarrollar una aplicación para un dispositivo que tuviera una versión de SO anterior a la 4.2 no tendríamos más remedio que desarrollarla en el JDE.

Una vez instalado el Plug-in en Eclipse y el componente con la versión del SO sobre la que se quiere desarrollar la aplicación tendremos acceso a todas las clases que forman parte de esa versión del SO, simuladores de los distintos dispositivos BlackBerry sobre los que poder realizar pruebas y también tendremos una pequeña utilidad que nos permite instalar nuestros códigos obtenidos de RIM para poder realizar la firma de los *cod* que usen clases que necesitan ser firmadas para su funcionamiento en la BlackBerry. Además, el Plug-in se encarga de la generación de los distintos archivos necesarios para el despliegue de la aplicación (*jad, alx, cod*) por lo que el desarrollador puede abstraerse de los pormenores sobre la generación de estos ficheros.

2.3.6 Simulador BlackBerry

RIM proporciona una serie de simuladores con los que se pueden realizar pruebas de las aplicaciones desarrolladas para comprobar cómo sería su comportamiento en el dispositivo antes de realizar su despliegue. En el caso del Plug-in para Eclipse, existen varios componentes correspondientes a las distintas versiones del SO de la BlackBerry. Estas versiones van desde la primera que se desarrollo para el Plug-in, la 4.2, hasta la más reciente que en estos momentos es la 5.0. Una cosa importante a tener en cuenta es que cada componente correspondiente a una versión tiene los simuladores de los dispositivos que usan esa versión de SO, por lo que si queremos probar nuestra aplicación en el modelo 83xx de la BlackBerry, deberemos instalar el componente cuya versión de SO es la 4.5, el cual incluye ese simulador (aparte de otros modelos más). Pero si lo que queremos es probarla en el modelo

Storm, entonces tendremos que hacer uso del componente cuya versión del SO es la 4.7, el cual incluye el simulador de la Storm.

Por defecto, a partir de la versión SO 4.2 las BlackBerrys realizan por defecto las conexiones de red usando el método BES/MDS. Por suerte tanto el JDE como el Plug-in incluyen un simulador MDS que permite simular las conexiones de red que el dispositivo podría hacer para comunicarse con cualquier otra aplicación que se encuentre en un servidor.

2.3.7 Consideraciones a tener en cuenta al desarrollar aplicaciones para BlackBerry.

Hay ciertos aspectos que un desarrollador debe tener en cuenta antes de iniciar un desarrollo de una aplicación para BlackBerry, y son las siguientes:

- *CPU y Memoria Limitada:* Tanto la CPU como la memoria RAM de la BlackBerry están bastante por debajo de las capacidades que pueden ofrecer un PC de sobremesa o un servidor. Existen muchas razones para estas limitaciones, entre las que podemos incluir la mayor duración de vida de la batería o el tamaño del dispositivo, pero básicamente lo que hay que tener en mente es que una aplicación que sea agresiva en el uso de CPU y RAM no va a poder ejecutarse correctamente en dispositivos BlackBerry, por lo que siempre hay que intentar evitar ejecutar los algoritmos agresivos en CPU en la BlackBerry. Además, debido a que el SO de BlackBerry es multitarea, aplicaciones que sean muy agresivas en CPU y RAM podrían perjudicar el buen funcionamiento de otras aplicaciones que estén corriendo en la BlackBerry.

- *Java como API nativa:* La máquina virtual de Java (VM) en BlackBerry envuelve todo el hardware al que la aplicación puede acceder. Esto quiere decir que la aplicación siempre estará en un entorno donde la VM interpreta los bytecodes, y no tiene acceso en tiempo real al hardware del dispositivo.

- *Tamaño limitado de pantalla:* El tamaño mayor de pantalla de una BlackBerry es de 360 x 480 pixeles de resolución, correspondiente a la Storm, por lo que el diseño de las UI deberán realizarse teniendo en mente estas restricciones.

- *Entrada del Usuario:* La BlackBerry Storm permite al usuario hacer click en cualquier punto de la pantalla, por lo que su comportamiento podría semejarse al uso del ratón de un ordenador. Sin embargo, el resto de dispositivos hacen uso de un trackball(rueda de desplazamiento), u otros más recientes de un trackpad, y teclado. El trackball tiene un comportamiento parecido a las flechas de dirección del teclado, por lo que los desplazamientos que pueden realizarse en estos dispositivos

son: arriba, abajo, derecha e izquierda. La UI deberá ser diseñada con esto en mente.

- *Muchos dispositivos diferentes:* Muchos tipos diferentes de dispositivos de BlackBerry se encuentran hoy día en el mercado, y entre las diferencias existentes entre ellos como puede ser la CPU o RAM, hay que destacar el tamaño de pantalla de los mismos, que puede variar desde los 240 x 260 pixeles de resolución a los 480 x 360, y también el tamaño físico del mismo. Otras diferencias pueden ser la entrada de usuario, como se ha comentado antes, o que, por ejemplo, algunos dispositivos tengan características hardware que otros no tienen, como por ejemplo el GPS. Se deberán tener en cuenta todos estos aspectos a la hora de diseñar la aplicación para que pueda abarcar el mayor número de usuarios posibles.

2.3.8 Patrones de Desarrollo Software Usados

Para el desarrollo de la aplicación se han usado los siguientes patrones de desarrollo de software:

- **Modelo-Vista-Controlador**: Este patrón es un estilo de arquitectura del software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. Este patrón es usado habitualmente en aplicaciones web, aunque también puede ser usado para aplicaciones de escritorio ó, como en este caso, para aplicaciones de dispositivos móviles. El patrón queda descrito por tres componentes:
    - *Modelo*: Es la representación específica de la información con la cual el sistema opera. Es recomendable usar patrones que abstraigan al sistema del modo en que se almacena esa información.
    - *Vista*: Representa el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.
    - *Controlador*: Responde a eventos, usualmente acciones del usuario, e invoca peticiones al modelo y, probablemente, a la vista.
- **DAO**: Este patrón es un componente software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de Datos o un fichero. La ventaja de usar el patrón DAO es que cualquier objeto de negocio no requiere conocimiento directo del destino final de la información que

manipula. DAO puede usarse para aislar a una aplicación de la tecnología de persistencia subyacente, lo que significa que podría ser actualizada o substituida sin realizar cambios en otras partes de la aplicación.

- **Singleton**: Este patrón está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso único o cuando cierto tipo de datos debe estar disponibles para todos los demás objetos de la aplicación. El patrón Singleton provee una única instancia global gracias a que:

    - La propia clase es la encargada de crear la única instancia.

    - Permite el acceso global a dicha instancia mediante un método de clase.

    - Declara el constructor de clase como privado para que no sea invocado directamente.

- **Template Method**: Es un patrón de diseño que define una estructura algorítmica en su súper clase, delegando la implementación a las subclases. Es decir, define una serie de pasos que serán redefinidos en las subclases. Usando Template Method se define una estructura de herencia en la cual la superclase sirve de plantilla de los métodos de las subclases. Una de las ventajas de este método es que evita la repetición de código y, por tanto, la aparición de errores.

- **Observer**: Es un patrón que define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, el observador se encarga de notificar este cambio a todos los otros dependientes. Todos los subscritores que deseen registrarse en el observador deben implementar unos métodos determinados mediante los cuales el observador es capaz de notificar a sus subscritores los cambios que se producen para que ellos puedan actuar en consecuencia.

3 DISEÑO Y RESOLUCIÓN DEL TRABAJO

Tal y como se ha comentado anteriormente, se utilizara el Proceso Unificado como guía para el desarrollo de la aplicación. En este apartado se describen los pasos realizados en las distintas fases del proceso, indicando el trabajo realizado en cada una de ellas.

3.1 Fase de inicio

El propósito de esta fase es establecer una visión común inicial de los objetivos del proyecto, determinar si es viable y decidir si merece la pena llevar a cabo algunas investigaciones serias en la fase de elaboración.

Respecto a los objetivos del proyecto, los mismos ya fueron enunciados en la sección 2.1. Una vez contemplado cuales son los objetivos que deseamos conseguir, lo siguiente que debemos comprobar es si es viable conseguirlo. Para ello se siguieron los siguientes pasos:

- *Comprobar API de BlackBerry y SugarCRM:*

  Ya que lo que deseamos es implementar una aplicación para un dispositivo móvil BlackBerry, la primera cuestión a la que debemos responder es: ¿se pueden desarrollar aplicaciones para los dispositivos BlackBerry? Por suerte, los ingenieros de RIM (empresa creadora de las BlackBerry) han implementado un sistema operativo basado en J2ME, con una API para que los desarrolladores puedan acceder a toda la funcionalidad proporcionada por la BlackBerry, como por ejemplo: almacenar datos en memoria persistente, realizar conexiones de red, comprobar la cobertura del dispositivo, cantidad de batería que le queda, creación de interfaces de usuario, etc.
  La segunda cuestión sería: ¿ofrece la aplicación SugarCRM una interfaz que te permita interactuar con ella desde el dispositivo BlackBerry? Como hemos comentado anteriormente, SugarCRM ofrece un punto de entrada a través de los Servicios Web ofrecidos por la aplicación, a partir de los cuales se puede comunicar cualquier aplicación desarrollada con SugarCRM.

  Por lo tanto, ya sabemos que es posible desarrollar una aplicación para los dispositivos móviles BlackBerry y que puedan interactuar con la aplicación SugarCRM a través de los Servicios Web ofrecidos por ésta.

- *Buscar aplicaciones que realicen la misma funcionalidad:*

  El segundo paso realizado dentro de esta fase inicial fue comprobar si existían otras aplicaciones que realizaran la misma funcionalidad que la

que deseamos implementar para nuestra aplicación. De aquí podemos obtener, por un lado, que verdaderamente se puede implementar la aplicación (ya que si otros lo han hecho, ¿por qué nosotros no?) y, por otro lado, se puede evaluar esa aplicación para poder "copiar" las virtudes de esa aplicación, y mejorar en aquellas cuestiones donde ésta falle. Haciendo una búsqueda por la red se encuentran dos aplicaciones que realizan justo lo que nosotros deseamos implementar, estas aplicaciones se llaman "Mobile Edge" y "Field Force".
En la Figura 2 se muestran algunas capturas de pantalla de la segunda aplicación:

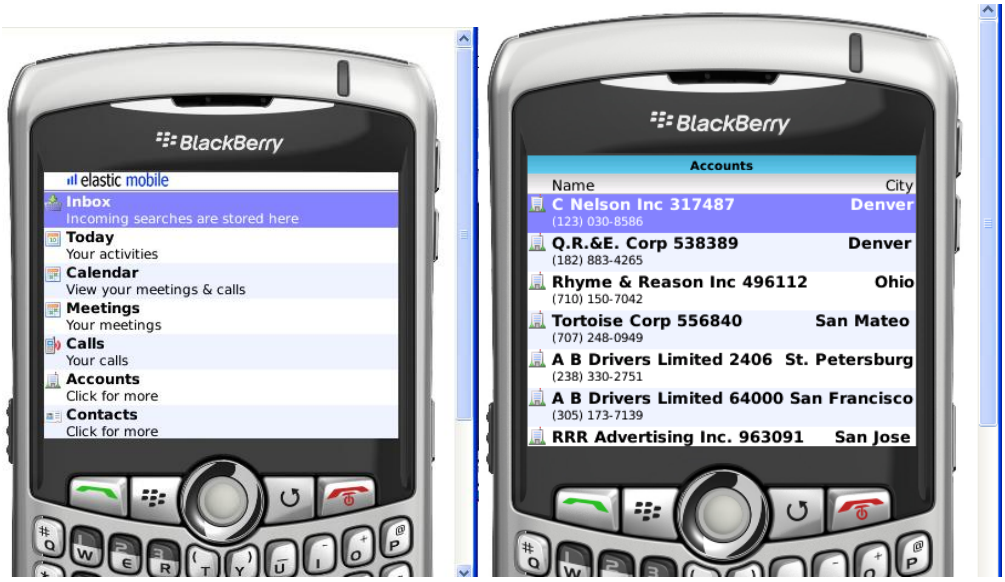

Figura 2. Interfaces de la aplicación Fieldforce.

Por lo tanto ya tenemos una base sobre la que empezar a trabajar. A continuación, lo que se hizo fue planificar la primera iteración de la fase de elaboración. Esta primera iteración consistirá en investigar sobre la API de BlackBerry (Rikz, 2009), ya que es necesario conocerla a fondo para poder desarrollar la aplicación. La duración de esta primera iteración es de 1 mes, puede que resulte algo excesivo, pero debemos tener en cuenta que hasta la fecha nunca hemos realizado una aplicación para un dispositivo móvil, por lo que estudiar y aprender todas las peculiaridades del desarrollo de aplicaciones para BlackBerry nos supondrá un esfuerzo que consumirá una buena parte del tiempo del desarrollo de la aplicación.

3.2 Fase de Elaboración

3.2.1 1ª Iteración (Duración - 1 mes):

En la fase inicial se estableció el trabajo que se realizara durante esta primera iteración. Para ello el estudio de la API se puede dividir en cuatro partes:

3.2.1.1 Estudio de las clases necesarias para la creación de una aplicación para el dispositivo.

Lo primero a destacar es el ciclo de vida de una aplicación ejecutada en un dispositivo BlackBerry. Este ciclo de vida es muy parecido a cualquier aplicación Java SE desarrollada. Por ejemplo, el método "*main*" desde el que se inicia la ejecución de la aplicación es idéntico para ambas. Mientras que pueden existir excepciones para aplicaciones específicas, la mayoría de aplicaciones para BlackBerry seguirán el mismo ciclo de vida, el cual se puede resumir en 5 pasos:

I. *Inicio de la aplicación:*

Una aplicación generalmente se inicia de alguno de los siguientes modos:

- El usuario hace "click" en el icono de la aplicación desde la ventana principal de la BlackBerry.
- La aplicación está preparada para arrancar automáticamente y es inicializada cada vez que arranca el dispositivo o es reiniciado.
- La aplicación es lanzada por otra aplicación.

En todos los casos, el método "*main*" es el punto de entrada para la aplicación. El dispositivo BlackBerry creará un proceso desde el que se llamará a este método. Cuando el método "*main*" termine, el proceso será terminado y la aplicación terminará de ejecutarse. Por lo tanto, cualquier cosa que deseemos hacer deberemos realizarlo antes de que finalice el método "*main*". Este método toma como parametros un array de objetos "*String*", que puede estar vacío (cosa que sucederá en la mayoría de los casos) o contener valores que pueden ser definidos tanto por el desarrollador desde las propiedades del proyecto, como por otro proceso que inicie la aplicación.

II. *Creación de la aplicación:*

Todas las aplicaciones BlackBerry que deseen presentar una interfaz de usuario al usuario deben extender de la clase "*UiApplication*". Solo se puede crear una instancia de esta clase para cada proceso, lanzando una excepción en tiempo de ejecución la maquina virtual si se intenta realizar una segunda instancia de la clase. Cualquier aplicación que no desee ofrecer una interfaz gráfica debe extender de la clase "*Application*". Desde cualquier punto de nuestra aplicación se puede ejecutar el método estático "*UiApplicaction.getUiApplication()*", el cual nos devolverá la instancia de nuestra aplicación.

III. *Invocación del hilo de evento (Event Thread):*

El hilo de evento lo inicia el sistema operativo del dispositivo BlackBerry por ti, pero éste no comienza a procesar los eventos y el dibujado de la interfaces gráficas hasta que explícitamente se lo pidamos. Para hacer esto debemos ejecutar el método "*UiApplication.enterEventDispacher()*". Una vez que se ejecuta este método, el hilo que ejecutó el método "*main*" pasa de estar bajo control del desarrollador a realizar la tarea de escuchar los eventos que puedan producirse por el usuario de la aplicación a través de las interfaces de usuario, así como del dibujado de las mismas en la pantalla del dispositivo. Aún se podrá ejecutar código creado por el desarrollador en este hilo, pero a partir de este momento será programado por el sistema operativo. Comentar que la llamada a este método no tiene retorno durante el resto de vida de la aplicación, por lo que si es necesario algún tipo de inicialización de la aplicación debe realizarse antes de la llamada a este método.

IV. *Procesamiento de eventos:*

La aplicación actuará en consecuencia con los eventos producidos, ya se produzcan por el teclado, el "trackball", la pantalla táctil o incluso otros eventos como los producidos cuando el dispositivo recibe mensajes o llamadas.

V. *Saliendo de la aplicación:*

Generalmente, una aplicación BlackBerry termina cuando la última ventana es eliminada (normalmente debido al cierre explicito de la ventana por parte del usuario) de la pila donde están almacenadas (display stack). Otra forma de terminar la aplicación sería ejecutando el método "*System.exit()*", pero es recomendado evitar esto y terminar la aplicación cerrando todas sus ventanas ya que de esta manera se liberan correctamente los recursos usados por la aplicación.

A continuación se profundiza un poco más en lo referente al hilo de evento, ya que este suele ser origen de muchos errores de programación al desarrollar aplicaciones para los dispositivos BlackBerry. La API para las interfaces graficas de BlackBerry es "*single-threaded*" (o traducido al español: hilo único), lo que significa que todas las actualizaciones de la interfaces gráficas y eventos producidos son gestionadas por el mismo hilo, o más concretamente, debe ser realizado por el hilo que obtenga el cerrojo de evento para poder realizar esa acción de manera síncrona. Este cerrojo por defecto es obtenido por el

hilo de evento cada vez que tiene que realizar alguna acción. Esto implica dos cosas: por un lado, otros hilos que no sean el de evento no pueden actualizar directamente la interfaz gráfica sin adquirir explícitamente el cerrojo, ya que si lo hicieran el sistema operativo lanzaría una excepción en tiempo de ejecución; y por otro lado, si se realiza una operación demasiado larga mientras se está en posesión del cerrojo, la interfaz de usuario se quedará "congelada" hasta que finalice.
Existe un método que permite determinar si el código actual se está ejecutando en el hilo de evento o en otro hilo creado por nosotros mismos. El método es "*UiApplication.isEventDispatchThread()*". Éste método nos devuelve un booleano indicándonos si estamos en el hilo de evento o no. Si este método nos devolviera un valor de falso, entonces tendríamos que obtener el cerrojo de evento para poder realizar cualquier cambio en la interfaz. Esto puede obtenerse de dos maneras:

1. Usando los métodos "*UiApplication.invokeLater()*" o "*UiApplication.invokeAndWait()*", los cuales ejecutaran las operaciones que le indiquemos a través de un objeto "Runnable" que le pasamos por parámetro, diferenciándose ambos en que el primer método vuelve inmediatamente tras realizar la llamada, por lo que no es bloqueante, y el segundo vuelve cuando todo el código del objeto "Runnable" que le pasamos por parámetro es ejecutado, por lo que es bloqueante.

2. Sincronizando el objeto devuelto por el método "*UiApplication.getEventLock()*", asegurando que las modificaciones de la interfaz se realizan de manera segura. Generalmente se recomienda usar el primer método, para evitar problemas de interbloqueos.

Otro aspecto interesante es detectar cuándo una aplicación se encuentra en background o foreground. Por defecto, una aplicación que se encuentre en foreground puede ser enviada a background pulsando la tecla roja del dispositivo BlackBerry (usada también para finalizar una llamada), mostrándose la pantalla Home del dispositivo, o cambiando explícitamente de aplicación manteniendo pulsada la tecla "menú" del dispositivo y seleccionando otra. Otra forma de enviar una aplicación a background sería realizándolo directamente desde nuestro código ejecutando el método "*UiApplication.requestBackground*". Para que una aplicación se dé cuenta de que está en background o foreground debe redefinir los métodos "*UiApplication.deactivate*" y "*UiApplication.activate*", los cuales serán llamados por el sistema operativo cada vez que esa aplicación sea enviada a background o foreground, respectivamente.

3.2.1.2 Estudio de las clases para la conexión en red del dispositivo.

En este apartado se estudia la API de BlackBerry que permite realizar conexiones de red a través de la red inalámbrica de datos. Lo primero que se debe examinar son los diferentes mecanismos en los que un dispositivo BlackBerry puede conectarse a la red inalámbrica, a saber: BlackBerry Enterprise Server/BlackBerry Mobil Data System (BES/MDS), BlackBerry Internet Service (BIS), direct Transmission Control Protoco/Internet Protocol (TCP/IP), WiFi, Wireless Access Protocol (WAP) 1.0 and WAP 2.0. Cada técnica tiene sus ventajas y desventajas, y dependiendo de la configuración del dispositivo y del entorno en el que se encuentre, algunas de ellas no estarán disponibles. Generalmente, se puede realizar cualquier tipo de conexión de red, como conexiones HTTP o sockets TCP o UDP, sobre ellos. A continuación, se ofrece una breve explicación de cada uno:

- *BES/MDS*:

  El BES permite a las BlackBerrys realizar conexiones seguras a servidores que se encuentren dentro de una red corporativa. Esto se logra a través del MDS del BES por lo que las conexiones creadas de esta forma suelen ser conocidas como conexiones BES/MDS. El MDS realiza la función de proxy entre el dispositivo BlackBerry y el servidor, es decir, el MDS realiza las conexiones en nombre del dispositivo, y los datos son transferidos hacia y desde el dispositivo por el mismo canal seguro que la corporación usa para el servicio de email de BlackBerry. Lógicamente, el método de conexión a través del BES/MDS solo es posible para dispositivos BlackBerry que hayan sido activados en el BES.
  La ventaja de usar el BES/MDS es que debido a que el MDS es quién verdaderamente realiza la conexión con el servidor, el dispositivo puede acceder a cualquier servidor que pueda resolver el MDS desde el servidor en el que se esté ejecutando el BES, de forma que todos los servidores que hayan detrás del firewall serán alcanzables por el dispositivo. La desventaja es que si existiera alguna restricción en el firewall a la hora de realizar conexiones a servidores que se encuentren fuera de la red corporativa, los dispositivos BlackBerry también se verían afectados por estas restricciones.

- *BIS*:

  BIS provee a usuarios finales mucha de la funcionalidad que proporcionada un BES a su red corporativa, pero sin el mismo nivel de seguridad. Si el dispositivo no está asociado a un BES, entonces está usando conexiones BIS para recibir y enviar correo. Además, el

BIS también realiza la función de proxy de la misma manera que lo hace el BES/MDS. La principal ventaja de usar BIS sobre una conexión directa sobre TCP/IP es que la mayoría de planes de datos no incluyen acceso a través de conexiones directas TCP/IP (por lo que nos podríamos llevar un buen susto en la factura a final de mes), sin embargo si incluyen el servicio BIS, ya que es necesario para recibir los emails si el dispositivo no está asociado a un BES. Para que una aplicación pueda realizar conexiones a través del BIS es necesario, en primer lugar, formar parte del Programa de Alianzas de BlackBerry y recibir la aprobación para que una aplicación pueda usar BIS.

- *Direct TCP/IP:*

  Al igual que la mayoría de los dispositivos móviles, los dispositivos BlackBerry pueden hacer uso de la infraestructura proporcionada por los proveedores de servicios para acceder a Internet sin tener que pasar a través de un servicio específico de BlackBerry (BES o BIS). La ventaja de este método es que está disponible para la mayoría de dispositivos, pero tiene dos inconvenientes principales:

  1. Se debe configurar el dispositivo para realizar estas conexiones, concretamente lo que se debe configurar es el APN (Access Point Name), por lo que si no está bien configurado las conexiones no se podrán realizar.

  2. Por otro lado, los planes de datos no suelen incluir este tipo de conexiones, que suelen ir tarificadas aparte.

- *WiFi:*

  Algunos dispositivos BlackBerry incluyen WiFi, lo que les permite acceder a Internet a través de un router WiFi. Las ventajas del uso de WiFi vienen en forma de mayor velocidad, baja latencia y por supuesto tu proveedor no te cobrará por el uso de la misma. Las desventajas están en la obligación de configuración de la WiFi para poder usarla, así como que la WiFi no está pensada para dispositivos móviles, en el sentido de que el radio de cobertura de la misma no abarca muchos metros, por lo que el usuario debería hacer uso de otro tipo de conexión para poder estar permanentemente conectado mientras se desplaza.

- *WAP 2.0:*

  WAP 2.0 hace uso de los Gateway WAP del proveedor. Similar a las conexiones directas TCP/IP, WAP 2.0 no usa ninguna infraestructura específica de BlackBerry. Por lo general todas las compañías proveedoras de servicios soportan este tipo de conexiones. La gran

ventaja de WAP 2.0 sobre las conexiones directas TCP/IP es que el primero no requiere de configuración por parte del usuario, ya que los dispositivos están creados para que automáticamente puedan usar ese tipo de conexión sin configuración adicional. La gran desventaja es que este tipo de conexiones no suelen ir incluidos en el plan de datos BlackBerry, por lo que su uso no está incluida en la tarifa plana, lo que supone una tarificación aparte.

- *WAP 1.0:*

  Al contrario que los mecanismos mencionados anteriormente para realizar una conexión a la red, WAP 1.0 no soporta todo el rango de conexiones disponibles. Concretamente, la seguridad es limitada. Por lo tanto, a menos que se tenga alguna necesidad excepcional, no se debería hacer uso de esta forma de conexión.

  En la Figura 3 se puede ver un esquema de las distintas formas de conexión a la red:

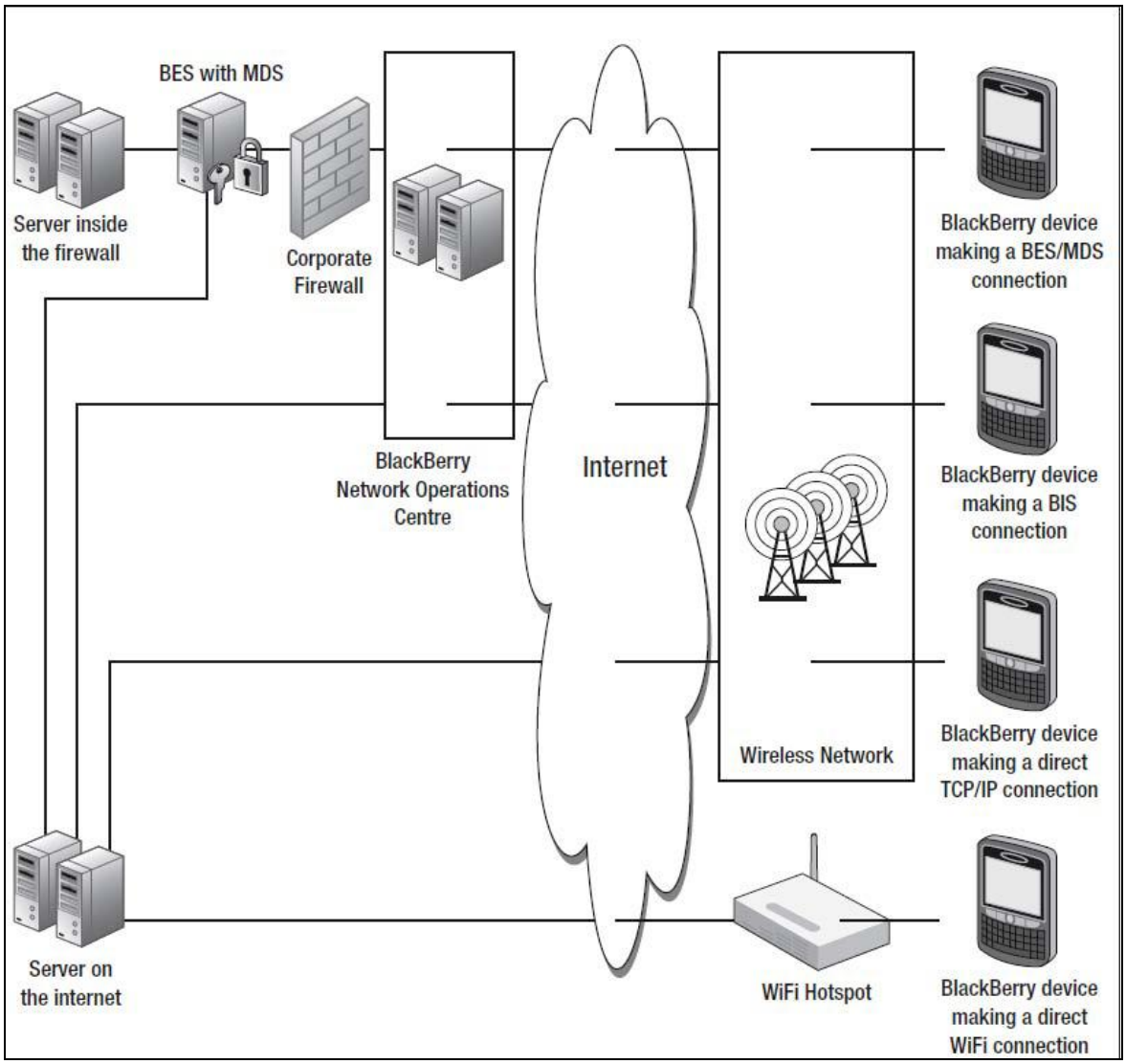

Figura 3. Diferentes formas de conexión a red desde un dispositivo BlackBerry.

Un aspecto adicional a tener en cuenta en el estudio de las clases para la conexión en red del dispositivo es el del denominado "Libro de Servicios" de los dispositivos BlackBerry. El "Libro de Servicios" se trata de un sistema de almacenamiento de configuración que el dispositivo usa para mantener y gestionar la información sobre distintos aspectos de su configuración. El libro de servicios contiene registros que abarcan desde la configuración de aplicaciones de terceros hasta la configuración de las cuentas de correo ligadas al dispositivo. Además, también almacena información sobre los distintos métodos en los que el dispositivo puede realizar una conexión a Internet, por lo que puede ser una manera útil y rápida de saber cómo puede salir el dispositivo a Internet. La única excepción sería para las conexiones directas TCP/IP, las cuales se pueden consultar y modificar en las opciones TCP/IP del dispositivo. Cada registro contiene dos identificadores, el CID y el UID. El UID identifica unívocamente el registro en el dispositivo, mientras que el CID da información sobre el tipo de registro que es. Por ejemplo, en un dispositivo que tenga asociadas varias cuentas de correo, habrán muchos

registros con el CID de CMIME, pero cada uno tendrá un diferente UID. No es aconsejable modificar el libro de servicio a menos de que se sepa con certeza qué se está realizando, ya que el dispositivo podría dejar de funcionar correctamente.

Una vez analizados los diferentes mecanismos de conexión y dónde se almacenan su configuración, a continuación se examina la clase que nos permitirá crear una conexión a un servidor. BlackBerry usa el mismo framework que se define en el estándar MIDP, con alguna funcionalidad extra específica de la BlackBerry. Todas las conexiones se inician usando la clase "*Connector*" (King, 2009). Esta clase puede crear diferentes tipos de conexiones (HTTP, HTTPS, socket, file, etc.) según la url que se le pase a su método "*Connector.open()*". Por ejemplo, para abrir una conexión HTTP para recibir una página Web se usaría lo siguiente:

```
HttpConnection connection = (HttpConnection)Connector.open("http://www.um.es");
```

La forma general de la url pasada por parámetro es la siguiente:

```
{esquema}://{objetivo}?{parametros}
```

Donde:

- *{esquema}* corresponde al nombre del protocolo, como por ejemplo http.
- {*objetivo*} es normalmente algo como la dirección web del recurso, o la ruta en el sistema de ficheros.
- {*parametros*} están formados como una serie de igualdades de la forma ";x=y". Por ejemplo, ";type=a".

Los protocolos que se soportan actualmente son: comm, socket, udp, sms, mms, http, https, tls o ssl y Bluetooth Serial Port Profile. Cada uno de los protocolos pueden tener sus parámetros específicos, pero para los protocolos encargados de comunicarse por la red existen dos parámetros generales para todos que nos ayudan a indicarle al dispositivo qué método de conexión debe usar. Estos parámetros son:

- *deviceside*: Especifica cuándo la conexión se realiza a través de BES/MDS o una conexión directa TCP/IP. Los valores posibles para este parámetro son "true" (TCP/IP) o "false" (BES/MDS).
- *interface*: Si se añade ";interface=wifi" al final de la URI, se abrirá una conexión WiFi.

Además de esta clase, existen otras clases auxiliares que facilitan la obtención de información sobre la cobertura existente, así como de los distintos tipos de método de conexión que existen.
La primera de ellas corresponde a "*CoverageInfo*", que permite determinar qué métodos de conexión están actualmente disponibles en el dispositivo BlackBerry. Para ello comprueba la radio del dispositivo, la cobertura de red y el libro de servicios para facilitar al usuario la información necesaria sobre los distintos métodos de conexión disponibles. El principal método para determinar la cobertura es "*CoverageInfo.getCoverageStatus()*", que devuelve una máscara de bits indicando los diferentes métodos de conexión disponibles, así como todas los tipos de redes disponibles, normalmente las redes móviles y WiFi. Además indica si pueden realizarse conexiones Bluetooth o USB hacia un ordenador. Por ejemplo la llamada a este método en un área con cobertura en un dispositivo que se encuentra activado en un BES y con un servicio de plan de datos que permite las conexiones directas TCP/IP, devolvería un mascara de bits que contendría el valor "*COVERAGE_MDS* | *COVERAGE_DIRECT* | *COVERAGE_BIS_B*", donde:

- *COVERAGE_MDS:* indica que pueden realizarse conexiones a través de un BES/MDS.
- *COVERAGE_DIRECT:* significa que pueden realizarse conexiones directas TCP/IP o WAP.
- *COVERAGE_BIS_B:* significa que pueden hacerse conexiones usando BIS.

La segunda clase corresponde a "*WLANInfo*", que permite comprobar si es posible realizar una conexión a través de WiFi. La llamada al método "*WLANInfo.getWLANState()*" devuelve el valor "*WLANInfo.WLAN_STATE_CONNECTED*" que nos indica que la WiFi está actualmente activada y conectada a una red, por lo que será posible realizar las conexiones WiFi; en caso contrario nos devolverá el valor "*WLANInfo.WLAN_STATE_DISCONNECTED*".
Notad que la clase anterior indica si el dispositivo posee WiFi, mientras que esta indica si la WiFi del dispositivo está correctamente configurada.

3.2.1.3 Estudio de las clases para el almacenamiento de datos en la memoria persistente del dispositivo.

Las BlackBerry, al igual que muchos otros dispositivos, utilizan memoria flash para almacenar datos de manera persistente, de forma que los datos puedan sobrevivir a reinicios del dispositivo o de la propia aplicación. Los dispositivos BlackBerry tienen una memoria flash interna, y muchos modelos también soportan una tarjeta externa SD. Algunos tipos de persistencia solo son aplicables a la memoria interna del

dispositivo, mientras que otros son aplicables tanto a la interna como a las tarjetas SD añadidas al dispositivo. Las BlackBerry ofrecen diferentes formas de realizar la persistencia de datos en el disporitivo, que son:

- *MIDP's Record Management System (RMS)*:

  RMS es soportado por BlackBerry principalmente para mantener la compatibilidad con el estándar MIDP. Es simplemente un formato de base de datos no relacional que permite a una aplicación almacenar arrays de bytes. Hay un soporte mínimo para compartir datos entre las aplicaciones, y los datos de las aplicaciones están ligados a ellas, en el sentido de que si se borra la aplicación, los datos también se borrarán. Generalmente, no hay razón para usar este método de persistencia, a menos que se quiera desarrollar un MIDlet en vez de una aplicación BlackBerry CLDC.

- *BlackBerry Persistent Store*:

  El BlackBerry Persistent Store provee características similares a RMS. Sin embargo, ofrece un manera más fácil de almacenar un mayor rango de objetos y la capacidad de almacenar directamente instancias de clases que se definan directamente en una aplicación. También ofrece opcionalmente tanto compresión como seguridad, aunque a costa de un pequeño trabajo extra por parte de la aplicación. Por tanto, para la mayoría de aplicaciones que precisen la funcionalidad necesaria para almacenar datos persistentes en la memoria interna del dispositivo, esta sería la opción más indicada.

- *JSR 75 FileConnection support*:

  El FileConnection APIs permite acceder al sistema de ficheros de BlackBerry, tanto al de la memoria interna como al de cualquier memoria externa que se haya añadido al dipositivo. El sistema de fichero es donde la BlackBerry almacena fotos, videos, ficheros que se descargan desde el navegador, y cualquier adjunto guardado desde un email, y es generalmente accesible desde cualquier aplicación instalada en el dispositivo BlackBerry. Es un buen lugar donde almacenar ficheros grandes, como fotografías o documentos, especialmente si el usuario desea luego acceder a ellos de otra manera, como por ejemplo desde la aplicación BlackBerry Desktop Manager.

De los tres métodos vistos anteriormente, el seleccionado en este proyecto para realizar la persistencia en el dispositivo es el segundo, que corresponde a BlackBerry Persistent Store. El motivo de esta selección es que la aplicación a desarrollar es una aplicación BlackBerry CLDC, por lo que no tendría sentido usar el método RMS para la persistencia, y por otro lado los objetos que se almacenan en memoria no son grandes, ni

tienen que ser accedidos, en principio, por otra aplicación, por lo que tampoco sería lógico hacer uso del FileConnection.

Como ya es conocido el BlackBerry Persisten Store permite almacenar datos en la memoria flash interna del dispositivo, y sólo en esta memoria. Sólo hay dos clases que debemos realmente conocer para poder hacer uso de este método de persistencia, y estas son: "*PersistentStore*" y "*PersistentObject*".

PersistentStore gestiona un lista de claves (cuyo valor es de tipo long) y objetos (instancias de PersistenObject). La lista de claves es global a todas las aplicaciones instaladas en el dispositivo. Desafortunadamente no es posible conocer a priori las claves que en un principio están en uso, pero en la práctica el rango es tan grande que la probabilidad de que coincidan dos claves es muy pequeña. Los dos métodos principales de esta clase son "*PersistentStore.getPersistentObject()*" y "*PersistentStore.destroyPersistentObject()*". A ambos métodos se les pasa la clave del objeto persistente que se quiere obtener o eliminar. Para el primer método, la función siempre devolverá una instancia de PersistentObejct, aunque la clave pasada no haya sido nunca usada para almacenar un objeto. Si esto sucediera, el contenido de este PersistentObject sería null, lo que indicaría que la clave no ha sido nunca usada, o que el objeto asignado a esta clave fue previamente eliminado. El segundo método sirve para eliminar el PersistentObject asociado a esa clave.

PersistentObject se encarga de contener los objetos que serán persistentes entre los reinicios del dispositivo o la aplicación. Una vez que se obtiene el "*PersistentObject*" a través de los métodos comentados anteriormente de la clase PersistentStore, se puede acceder al contenido de este objeto a través del método "*persistentObject.getContents()*". Es preciso recordar que el valor devuelto por este método podría ser null, por los motivos comentados anteriormente. Para establecer o reemplazar el contenido de este PersistentObject se usa el método "*persistentObject.setContents()*", donde se pasaría como parámetro el objeto que se desee almacenar. Aunque establecer el contenido no significa que automáticamente se almacene en memoria. Para que esto suceda se debe ejecutar el método "*persistentObject.commit()*". Después de la ejecución de este método el objeto habrá quedado correctamente almacenado en la memoria persistente. "*PersistentObject*" contiene una referencia a su contenido, por lo que si se desea modificar el contenido, lo único que hay que hacer es modificar la instancia del contenido y, después, ejecutar el método commit para que los cambios queden reflejados en la memoria. No sería necesario volver a ejecutar el método "*setContents*" a menos que lo que se desee sea almacenar otro objeto distinto.

Para terminar comentar que PersistentObject sólo puede contener objetos, por lo que no puede almacenar tipo primitivos. Por tanto, si se precisase almacenar, por ejemplo, un entero, deberíamos envolverlo dentro de un objeto Integer para después añadir éste al contenido del PersistentObject. Además, no todos los objetos pueden añadirse al contenido del PersistentObject, sino que sólo se podrán almacenar los objetos que implementen la interfaz "*Persistable*". Esta interfaz no contiene métodos, sólo es usada como una marca para el sistema operativo BlackBerry. Muchas de las clases pre-construidas para BlackBerry ya implementan esta interfaz, que pueden ser visualizadas en el Javadoc de la interfaz Persistable. Por lo tanto, si se intentara hacer persistente un objeto que no implemente esta interfaz, el sistema operativo lanzaría una excepción en tiempo de ejecución. Esto no ocurre, sin embargo, para algunas clases básicas de Java que pueden ser almacenadas a pesar de no implementar la interfaz Persistable, como: Boolean, Byte, Character, Integer, Long, Object, Short, String, Vector y Hashtable. Además, los arrays de los tipos primitivos y de las clases anteriormente comentadas también pueden almacenarse de forma persistente aún no implementando la interfaz Persistable.

Para terminar, comentar que existe otro tipo de almacenamiento en la memoria interna del dispositivo conocida como "Runtime Store". La principal diferencia con las anteriores es que no es persistente, por lo que todo lo que almacenemos aquí se borrará cuando el dispositivo se reinicie. Generalmente suele utilizarse para compartir información entre distintas aplicaciones de una forma eficiente.

3.2.1.4 Estudio de las clases para la generación de interfaces de usuarios.

La API BlackBerry incluye un completo framework para la construcción de interfaces de usuario para nuestras aplicaciones. La forma de crear interfaces de usuario en BlackBerry es muy parecida a como se hacen con otras herramientas como AWT Swing, Windows Forms, o cualquier otra herramienta de generación de interfaces gráficas orientada a objetos, por lo que si se tiene algo de experiencia en estas herramientas, no supondrá un gran esfuerzo adaptarse al framework de BlackBerry. El punto clave que diferencia a la generación de interfaces de usuario de BlackBerry de los anteriores ejemplos vistos es que la generación de las interfaces se hace completamente a través del código, no existen ficheros de configuración o metadatos externos sobre los que preocuparse. Esto tiene aspectos positivos, como por ejemplo que toda la información de la interfaz gráfica se encuentra centralizada en el código, y aspectos negativos, como que no se puede hacer uso de herramientas visuales para la construcción de interfaces gráficas.

A continuación se anuncian los distintos componentes existentes para crear interfaces gráficas en BlackBerry. Todos los elementos visibles en una ventana de una aplicación son de uno de los siguientes tipos:

- *Fields*: Este es el bloque de construcción básico de la interfaz gráfica. Todos los controles existentes corresponden a una instancia de un field. Existen algunos fields pre-construidos, como los botones o los campos editables, que ofrecen una funcionalidad básica para poder construir aplicaciones rápidamente, aunque el aspecto visual de los mismos es algo pobre. Los fields se encargan de dibujarse así mismos y de manejar las entradas del usuario.

- *Managers*: Son los encargados de "colocar" los fields en la ventana. Cada field solo puede estar ligado a un Manager, lanzando el sistema operativo una excepción en caso de que estuviera ligado a más de un manager. Cada manager es un Field, lo que significa que un manager puede contener otros managers, permitiendo crear complejos posicionamientos de los fields en la ventana.

- *Screens*: Hay una ventana activa por aplicación en cualquier momento. Las ventanas manejan los fields incluidos en ella a través de un manager llamado Manager Delegado, y provee funcionalidad adicional, como por ejemplo los menús de la aplicación.

Todos los fields son derivados de la clase "*Field*". Los fields que ya se encuentran pre-construidos se pueden encontrar en el paquete "*net.rim.device.api.ui.component*". Todos los managers derivan de la clase "*Manager*", la cual también deriva de la clase "*Field*". Los managers que ya se encuentran pre-construidos se pueden encontrar en el paquete "*net.rim.device.api.ui.container*". Todas las ventanas derivan de la clase "*Screen*", la cual también deriva de la clase "*Manager*", y por lo tanto también de la clase "*Field*". Las ventanas pre-construidas también pueden encontrarse en el paquete "*net.rim.device.api.ui.container*". Es preciso resaltar una excepción que se producen con las ventanas, y es que el hecho de que hereden de la clase "*Field*" solo significa que comparten funcionalidad, es decir, que un "*Screen*" hace cosas que hace un "*Field*", como pintarse a sí mismo, manejar las entradas del usuario, y gestionar las campos que estén incluidos en ella, pero no se podría añadir una ventana a un manager, es decir, no es posible substituir un "*Screen*" por un "*Field*" de la forma en que se puede substituir un "*Manager*" por un "*Field*".

En lo referente al manejo de las entradas del usuario, la API de BlackBerry utiliza el patrón "observer" para despachar los eventos. Todos los fields pueden tener un "listener", al que se le notificará cada vez que se produzca un cambio en el estado del field debido al evento ocurrido, para que el listener actúe en consecuencia. Los eventos que produzcan un cambio de estado en un field variaran de un field a otro. Por ejemplo,

en un "*ButtonField*" el evento se lanza cuando el botón es pulsado por el usuario, en cambio, en un "*CheckBoxField*" el evento se lanza cuando el campo se marca o desmarca. Para añadir un listener a un Field se hace uso del método "*Field.setChangeListener()*", al cual se le pasa como parámetro el listener deseado. Notar que este modelo sólo permite tener un listener por field, o ninguno. Cada vez que se haga una llamada al método para establecer el listener, este reemplazará al anterior, si es que existía. Un listener debe implementar la interfaz "*FieldChangedListener*", que contiene el método que es llamado por el field cada vez que se produce un cambio en su estado.

Tal y como se comentó previamente, existen una serie de clases pre-construidas que pueden usarse para generar nuestras interfaces de usuarios, aunque también es posible crear componentes propios, que permitirán enriquecer las interfaces de usuario de las aplicaciones. Para generar un componente propio, el mínimo número de métodos que es necesario redefinir de la clase Field son los métodos paint y layout (o sublayout, si se está creando un manager). Esto se debe a que el funcionamiento del dibujado de cualquier componente que se encuentre en una ventana transcurre básicamente en dos etapas:

1) *Diseño de los componentes en la pantalla*:

   Esta etapa envuelve el posicionamiento y dimensionado de los managers y fields que se encuentren en la pantalla. El trabajo comienza desde la propia ventana y va descendiendo hacia todos los managers y fields contenidos de la siguiente forma:

   i. El método "*Screen.sublayout()*" de la ventana es invocado. A este método se le pasa el ancho y alto de la pantalla del dispositivo en pixeles.

   ii. El método sublayout del manager delegado de la ventana es llamado. Como se ha comentado, el manager delegado es el componente que contiene todos los fields y managers añadidos a la ventana, y es el único componente controlado directamente por la ventana. El manager delegado generalmente usará todo el ancho y alto disponible que se le pase, pero en algunos tipos de ventanas esto no sucede así, por lo que el ancho y alto usado por el manager delegado será menor que el que la ventana indique.

   iii. El manager delegado iterará por todos los fields y managers que contenga y les pedirá que se dimensionen a sí mismos ejecutando los métodos layout (en el caso de los fields) y sublayout (en el caso de los managers), obteniendo así la posición y tamaño deseado de cada componente. El ancho y alto disponible para los fields y los managers variará dependiendo de cómo el manager delegado realice la disposición de sus componentes. En algunos

casos, podría ser mayor que el ancho y alto que el propio manager delegado tiene, lo que significaría que este manager realizará scrolling y solo dibujará un subconjunto de los campos incluidos en un momento determinado.

iv. Cada manager llamará a sus managers, fields y así sucesivamente para que cada uno quede posicionado en la ventana.

v. Cada field se diseñará a sí mismo, pudiendo utilizar menos tamaño que el pasado por su manager. Si el field usara mas tamaño que el disponible, no daría ningún error al realizar el layout, pero a la hora de dibujarse solo se vería en la pantalla el trozo del field que estuviera dentro del tamaño máximo pasado por el manager.

2) Dibujado de los componentes de la pantalla:

La etapa de pintado es donde los píxeles se dibujan en la pantalla del dispositivo. En la misma secuencia que en el diseño, la ventana, managers y fields se invocan para que se pinten a sí mismos de la siguiente forma:

i. El método "Screen.paint()" es llamado, con un "*Graphics*" pasado por parámetro que representa la pantalla actual.

ii. La ventana puede pintar algo de ella misma, como por ejemplo el fondo, y entonces pedirle a su manager delegado que se pinte a sí mismo, lo que se consigue llamando al método paint del manager delegado pasándole como parámetro el mismo objeto Graphics. Si el tamaño de el manager delegado es menor que el de la pantalla, entonces el objeto Graphics se "recorta" al tamaño del manager, evitando de esta manera que el manager se dibuje en una posición de la pantalla que no le corresponde, y evita al manager preocuparse sobre cuál es la posición absoluta sobre la que empezar a pintarse, la cual, generalmente, desconoce.

iii. El manager delegado también pinta algo de sí mismo y pide a sus fields y managers contenidos que se pinten a sí mismos, estableciendo los "recortes" apropiados para cada uno.

iv. Cada manager se pinta a sí mismo y pide a sus managers y fields contenidos que hagan lo mismo.

v. Cada field se pinta a sí mismo.

Un punto importante a tener en cuenta es que la etapa de diseño sucede muy pocas veces a lo largo de la ejecución de la aplicación (generalmente cuando se crea una ventana o los fields son añadidos o

borrados), mientras que la etapa de pintado se realiza frecuentemente. Esto implica que las operaciones realizadas en la etapa de pintado deben ser muy eficientes, ya que si las mismas son lentas o muy pesadas, el usuario obtendrá una experiencia negativa a la hora de hacer uso de la aplicación.

Por último, hacer mención a la clase "Graphics" anteriormente comentada. Esta clase se le pasa como parámetro a los métodos paint de todos los componentes de una ventana para que puedan pintarse a través de ella en la pantalla. Es decir, esta clase da las herramientas necesarias para que se pueda pintar cualquier componente que se pueda ver en cualquier aplicación desarrollada para BlackBerry. Es necesario tener siempre en mente que la instancia de Graphics que se le pase a los métodos paint es siempre la misma, por lo que si en algún método se modifica algún atributo de esta instancia, esos cambios se verán reflejados en el resto de llamadas. Si, por ejemplo, queremos pintar un field de color naranja, y cambiamos la propiedad correspondiente al color de Graphics al color naranja, a partir de ese momento todos los fields y managers que pintemos se harán en color naranja, a menos que volvamos a cambiar el color.

Con esto, se da por concluida la primera iteración de la fase de elaboración. Comentar que mientras se realizaba el estudio de la API de BlackBerry, se fueron probando una serie de ejemplos que RIM provee a los desarrolladores donde se pueden ver el uso de las clases comentadas anteriormente. Para ello solo es necesario importar en Eclipse el espacio de trabajo que contiene varios proyectos donde se tocan diferentes aspectos sobre el desarrollo para BlackBerry, como son las interfaces gráficas, la conexión a red, el uso del GPS, el clásico "Hellow World!", etc.

A continuación, se planifica la 2ª iteración de esta fase de elaboración. Dado que en la primera iteración no se escribió código explícito para la aplicación, sino que se fueron probando los ejemplos proporcionados a los desarrolladores, no se pudo generar al final de la primera iteración un prototipo de la misma, a partir del cual ir incrementando y desarrollando nuevos aspectos de la misma. Para esta nueva iteración si se plantearán los objetivos necesarios para poder tener una primera versión ejecutable de la aplicación, ya que uno de los objetivos de cada iteración del proceso unificado es obtener una parte funcional de la aplicación final. Luego en esta segunda iteración se intentará abarcar un aspecto crucial en la aplicación, la comunicación entre SugarCRM y el dispositivo BlackBerry. Para ello se implementará la comunicación necesaria para realizar el login del usuario, así como la interfaz gráfica. Se debe recordar que un aspecto importante en esta etapa de elaboración es abordar los problemas más críticos que puedan surgir en el desarrollo de la aplicación.

3.2.2 2ª Iteración (Duración - 1 semana):

Tal y como se comentó previamente, la aplicación BlackBerry se comunicará con el servidor a través de los servicios web que ésta proporciona. Por lo tanto lo primero que se debería hacer es estudiar la API con los servicios ofrecidos por SugarCRM. Esto se puede hacer de dos maneras:

- *Desde un navegador web:* Desde cualquier navegador web (como FireFox o Internet Explorer) puede realizarse una llamada al servidor para que devuelva una página con todas las llamadas posibles que se pueden realizar al servidor a través de sus servicios web. Para ello basta con introducir en el navegador la url "*http://<hostname>/service/v2/soap.php*", y se obtiene la página mostrada en la Figura 4:

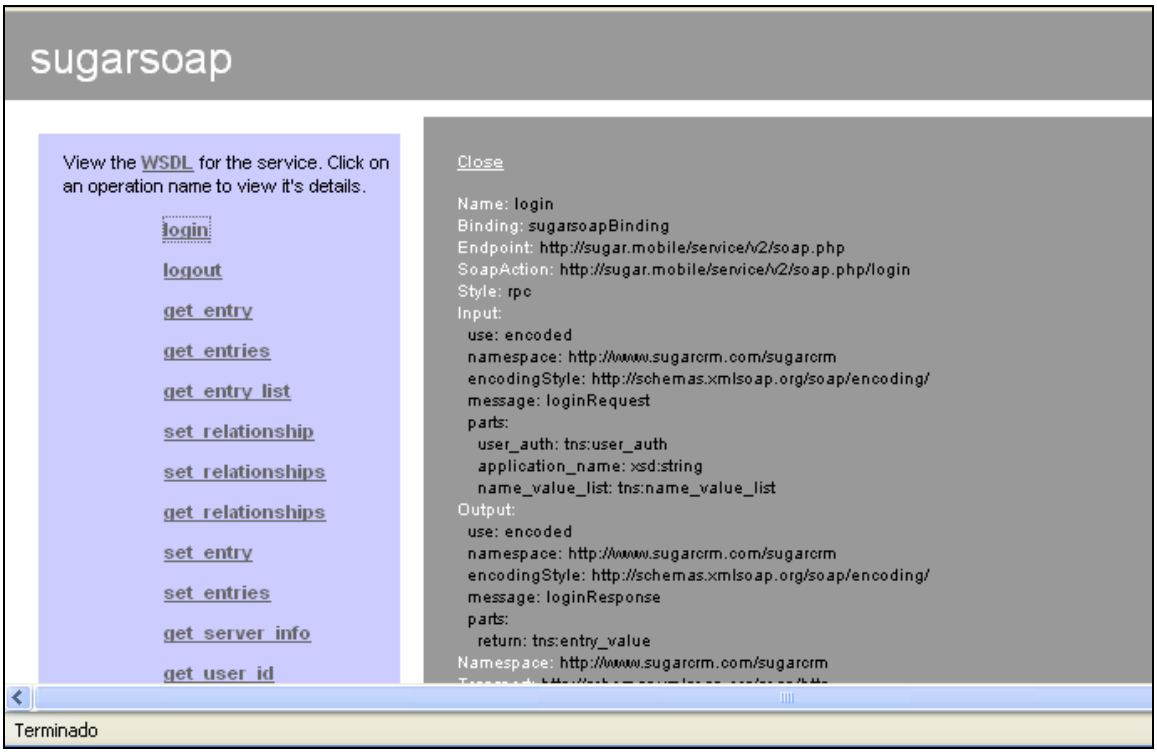

Figura 4. Servicios Web ofrecidos por SugarCRM

Desde aquí se pueden ver todas las funciones ofrecidas por SugarCRM, así como los parámetros de entrada y salida que usados en cada función, el punto de entrada al servicio web, etc. Como se puede ver, la función login recibe tres parámetros de entrada que son:

  - *user_auth*: Este parámetro es de tipo "user_auth". Para saber a que corresponde este tipo exactamente se puede pinchar en el enlace de la página correspondiente a WSDL y se nos mostraran todos los tipos de datos usados por en los servicios web de SugarCRM. El tipo "user_auth" corresponde a un tipo compuesto formado por:

- *user_name*: Corresponde al nombre del usuario que desea iniciar la sesión en SugarCRM. Es de tipo "string".

- *Password*: Corresponde al password del usuario que desea iniciar sesión. Es de tipo "string".

Se muestra un ejemplo de cómo se ve esta información en el navegador web:

```
- <xsd:complexType name="user_auth">
  - <xsd:all>
      <xsd:element name="user_name" type="xsd:string"/>
      <xsd:element name="password" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
```

- *application_name*: El nombre de la aplicación desde la que se está realizando el login. Actualmente no se hace uso de este parámetro. Es de tipo "string".

- *name_value_list*: Es un lista de pares nombre-valor que pueden ser usados como parámetros extras a la hora de realizar el login. La aplicación no hará uso de este parámetro inicialmente. La lista está formada por elementos compuestos de tipo "name_value", que a su vez están formados por dos strings correspondientes al nombre del elemento y al valor asignado al mismo. La definición de estos tipos compuestos es la siguiente:

```
- <xsd:complexType name="name_value">
  - <xsd:all>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
- <xsd:complexType name="name_value_list">
  - <xsd:complexContent>
    - <xsd:restriction base="SOAP-ENC:Array">
        <xsd:attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="tns:name_value[]"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
```

La función devuelve un valor de tipo "*entry_value*", que es un tipo compuesto definido de la siguiente manera:

```
- <xsd:complexType name="entry_value">
  - <xsd:all>
      <xsd:element name="id" type="xsd:string"/>
      <xsd:element name="module_name" type="xsd:string"/>
      <xsd:element name="name_value_list" type="tns:name_value_list"/>
    </xsd:all>
  </xsd:complexType>
```

Donde:

- *id*: Corresponde al identificador de la sesión actual creada al realizar correctamente el login. Este identificador es usado en las sucesivas llamadas que se hagan al servidor.

- *module_name*: El nombre correspondiente al módulo usuario, generalmente llamado "User".

- *name_value_list*: Mismo tipo que el comentado previamente en los parámetros de entrada de esta función.

- *Desde un editor de texto:* Si bien la forma anterior de conocer la interfaz de una de las funciones ofrecidas por el servicio web es la más rápida y agradable, existe un pequeño problema, y es que no se explica exactamente para que sirve cada parámetro. Para obtener esta información es necesario abrir el fichero php donde se encuentran implementadas las funciones del servicio web, y allí se encuentra la definición de cada parámetro, ya que todas las funciones desarrolladas están correctamente documentadas. En el caso de la función login se encuentra la siguiente documentación:

```
/**
 * Log the user into the application
 *
 * @param UserAuth array $user_auth -- Set user_name and password (password needs to be
 *      in the right encoding for the type of authentication the user is setup for.  For Base
 *      sugar validation, password is the MD5 sum of the plain text password.
 * @param String $application -- The name of the application you are logging in from.  (Currently unused).
 * @param array $name_value_list -- Array of name value pair of extra parameters. As of today only 'language' and 'n
 * @return Array - id - String id is the session_id of the session that was created.
 *               - module_name - String - module name of user
 *               - name_value_list - Array - The name value pair of user_id, user_name, user_language, user_currency_
 * @exception 'SoapFault' -- The SOAP error, if any
 */
public function login($user_auth, $application, $name_value_list){
    $GLOBALS['log']->info('Begin: SugarWebServiceImpl->login');
    global $sugar_config, $system_config;
    $error = new SoapError();
    $user = new User();
    $success = false;
    if(!empty($user_auth['encryption']) && $user_auth['encryption'] === 'PLAIN'){
        $user_auth['password'] = md5($user_auth['password']);
    }
```

De aquí en adelante, cuando se comenten las funciones usadas de los servicios web, se usará la primera forma para explicar los parámetros tanto de entrada como de salida que se usarán.

Tal y como se comentó previamente, para realizar las llamadas a los servicios web desde la BlackBerry se hará uso de la librería "*ksoap*". Esta librería contiene un conjunto de clases que permiten crear mensajes soap que pueden ser enviados y, posteriormente, permite leer los mensajes soap recibidos como respuesta. Como aún no se ha mostrado el uso de la librería, para la implementación de la función login se mostrarán los pasos dados para crear el mensaje soap, enviarlo al servidor y recibir la respuesta. Para el resto de funciones no se volverá a escribir estos pasos, ya que serán muy parecidos a los comentados para esta función, excepto por los parámetros que puedan tener cada función, por lo que sólo se añadirá el código de las clases encargadas de realizar el mapeo de los parámetros a los mensajes soap y viceversa.

Las clases encargadas de trabajar con la librería ksoap y enviar los mensajes soap al servidor a través del protocolo HTTP se encuentran en los paquetes:

- "*com.neosistec.sugarMobile.soap*": En este paquete se encuentran las clases *ControladorSoap.java*, ControladorCobertura.java, Bitacora.java y Accion.java. En este apartado tan se comenta la primera clase, que es la encargada de crear los mensajes SOAP usando la librería ksoap. El resto de clases se irán comentando conforme vayamos abarcando los problemas que resuelven las mismas.

- "*com.neosistec.sugarMobile.soap.sugarcrm*": En este paquete se encuentran las clases encargadas de mapear tanto los mensajes de respuesta obtenidos del servidor como los parámetros usados en los mensajes. Para el caso de la función login, se usan las clases LoginResponse.java, EntryValue.java, UserAuth.java y NameValue.java.

- "*com.neosistec.sugarMobile.soap.transport*": En este paquete se encuentran las clases encargadas de generar la petición HTTP que contendrá el mensaje SOAP que se desee enviar. Las clases usadas son HttpTransport y ServiceConnectionBlackBerry.

Toda clase que desee mapear los parámetros de un mensaje SOAP debe implementar la interfaz KvmSerializable, que contiene cuatro métodos que son usados por la librería tanto para inicializar el objeto con los datos obtenidos en la respuesta del mensaje SOAP, como para crear el mensaje de petición con los parámetros inicializados a los valores que nosotros deseemos. Esto solo es necesario realizarlo con los parámetros de tipos complejos, es decir, para los parámetros de tipo básico como enteros, strings, arrays, etc., no es necesario crear estas clases de mapeo, ya que la librería es capaz de mapearlos automáticamente. A continuación se muestra la implementación de la clase "*EntryValue.java*" usada por la

librería KSOAP para realizar los mapeos correspondiente al tipo de dato devuelto por la función login. Debido a la definición del tipo de dato "*entry_value*" mostrado previamente, la clase EntryValue.java encargada de mapear ese tipo de dato es definida de la siguiente manera:

```
public final class EntryValue implements IBaseObject {

      private String id;
      private String moduleName;
      private Vector nameValueList;
```

Donde IBaseOject es una interfaz que hereda de KvmSerializable y está definida de la siguiente manera:

```
public interface IBaseObject extends KvmSerializable
{
  public static final String NAMESPACE = "http://www.sugarcrm.com/sugarcrm";

  public void register(SoapSerializationEnvelope envelope);
}
```

El string "NAMESPACE" corresponde a la url donde se encuentran definidos todos los tipos usados por el servicio web de sugarcrm, y el método "register" es el encargado de añadir los mapeos de un tipo complejo del mensaje SOAP a la clase correspondiente. La clase "EntryValue" implementa el método "register" de la siguiente manera:

```
public void register(SoapSerializationEnvelope envelope)
{
  envelope.addMapping(NAMESPACE, "entry_value", this.getClass());
  new NameValue().register(envelope);
}
```

Como se puede apreciar, el método recibe por parámetro un objeto de tipo "*SoapSerializationEnvelope*" que es el encargado de parsear el mensaje SOAP, tanto el de la petición como el de la respuesta, y lo que se le indica a través del método "addMapping" es que para parsear el tipo complejo "entry_value" dentro del mensaje SOAP debe hacer uso de esta clase. Luego se crea un objeto de tipo NameValue, y se realiza una llamada a su método "*register*". Esto se debe a que uno de los atributos de la clase "*EntryValue*" corresponde a una lista de "*NameValue*", y como el tipo "*NameValue*" es un tipo complejo, hay que llamar a su método register y pasarle por parámetro el objeto al cual le especificará como debe realizar el parseo del tipo compuesto "*NameValue*". Se harán tantas llamadas a los métodos register como tipos complejos intervengan en la función concreta.

Los cuatro métodos implementados de la interfaz KvmSerializable son los siguientes:

```
public Object getProperty(int index)
{
  switch(index)
  {
    case 0:
      return id;

    case 1:
      return moduleName;

    case 2:
      return nameValueList;
    default:
      throw new RuntimeException("ErrorValue.getProperty: Indice
      incorrecto");
  }
}
```

El método "*getProperty*" es el encargado de devolver la propiedad según el índice especificado. Si se mira de nuevo la definición del tipo "*entry_value*", se ve cómo el primer elemento corresponde al "*id*", el segundo al "*module_name*" y el tercero al "*name_value_list*", por lo que en la implementación si se solicita el elemento correspondiente a la posición 0 (el primer elemento), se devuelve el valor del atributo "*id*", y así sucesivamente. Debido a que este tipo de dato complejo solo tiene tres elementos el valor máximo que podría tener el parámetro "index" sería de 2, por lo que si se pasa un valor mayor el método lanza una excepción indicando este error.

```
public int getPropertyCount()
{
  return 3;
}
```

El método "*getPropertyCount*" es el encargado de indicar el número de atributos que forman parte de este tipo complejo.

```
public void getPropertyInfo(int index, Hashtable properties, PropertyInfo
      info)
{
  info.type=PropertyInfo.STRING_CLASS;
  switch(index)
  {
    case 0:
      info.name="id";
      break;

    case 1:
      info.name="module_name";
      break;

    case 2:
```

```
        PropertyInfo infoItem=new PropertyInfo();
        infoItem.name="item";
        infoItem.type=new NameValue().getClass();
        info.name="name_value_list";
        info.type=PropertyInfo.VECTOR_CLASS;
        info.elementType=infoItem;
        break;

      default:
        throw new RuntimeException("ErrorValue.getProperty: Indice
        incorrecto");
    }
}
```

El método "*getPropertyInfo*" es el encargado de rellenar el objeto de tipo "*PropertyInfo*" pasado por parámetro con los datos correspondientes al índice pasado por parámetro. Así, si el índice es 0, se rellena el objeto indicando que el nombre del elemento correspondiente es "*id*", y que es de tipo "*String*", pero si el índice es 2 se rellena el objeto indicando que el nombre del elemento es "*name_value_list*", que es de tipo "*Array*" (la librería ksoap se encarga de traducir automáticamente, cuando realiza el parseo del mensaje SOAP, el tipo Vector por un tipo Array y viceversa), y además se indica que los elementos del array tendrán el nombre de "*item*", y serán de tipo "*NameValue*", es decir, se está definiendo el siguiente tipo complejo:

```
- <xsd:complexType name="name_value_list">
  - <xsd:complexContent>
    - <xsd:restriction base="SOAP-ENC:Array">
        <xsd:attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="tns:name_value[]"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
```

Y por último:

```
public void setProperty(int index, Object value)
{
  switch(index)
  {
    case 0:
      id=(String)value;
      break;

    case 1:
      moduleName=(String)value;
      break;

    case 2:
      nameValueList=(Vector)value;
      break;

    default:
      throw new RuntimeException("ErrorValue.getProperty: Indice
      incorrecto");
```

```
    }
}
```

El método "*setProperty*" es el encargado de establecer el atributo de la clase referida por el parámetro "*index*" con el valor correspondiente pasado por el parámetro "*value*".

El resto de clases que se usen para mapear los tipos complejos de los parámetros usados en la función login tendrán una implementación parecida a la clase EntryValue comentada, diferenciándose lógicamente en el número de atributos que pueda tener, así como en el tipo de datos de esos atributos.

A continuación se comenta el código implementado en la clase "*ControladorSOAP*", encargada de realizar el login con el servidor SugarCRM. El método de la clase encargado de realizar esta función es:

```
public int doLogin(String userName, String passwordMD5)
```

El método recibe dos cadenas como parámetros que se corresponden con el nombre de usuario que desea iniciar su sesión, y el password del mismo, que previamente ha sido encriptado usando el algoritmo de cifrado MD5. Una vez que este método es llamado y se le pasan los parámetros requeridos, se realizan las siguientes tareas para realizar el login del dispositivo BlackBerry con el servidor:

```
if(!controladorCobertura.hayCobertura())
{
  return resultadoLogin;
}
```

Primero se realiza una comprobación de la cobertura que dispone el dispositivo. Si no fuera la suficiente (o simplemente no la tenga), el método termina su ejecución indicando que no pudo realizarse el login por falta de cobertura. El encargado de comprobar la cobertura e indicar a las demás clases si en el momento actual existe cobertura es la clase "*ControladorCobertura*". Más adelante se comenta como realiza esta tarea.

```
UserAuth userAuth = new UserAuth(userName, passwordMD5/*, "1.0"*/);

PropertyInfo infoNameValues=new PropertyInfo();
PropertyInfo infoNameValuesItems=new PropertyInfo();
infoNameValues.name="name_value_list";
infoNameValues.type=PropertyInfo.VECTOR_CLASS;
infoNameValuesItems.name="item";
infoNameValuesItems.type=new NameValue().getClass();
infoNameValues.elementType=infoNameValuesItems;

Vector nameValues=new Vector();
```

A continuación viene la creación de los parámetros que se enviarán en el mensaje SOAP de la función login. Como se puede comprobar, de los tres parámetros que tiene la función login (que son "*user_auth*", "*application_name*" y "*name_value_list*") solo se han inicializado el primero y el último. Esto se debe a que el primer parámetro corresponde a un tipo de dato complejo, comentado previamente, por lo que es necesario crear una instancia de la clase "*UserAuth*", que será la encargada de realizar el mapeo de los valores de ese parámetro al mensaje SOAP. Como puede verse los valores con los que es inicializada la clase corresponden al nombre de usuario y al password que fueron pasados por parámetros al método. Respecto al tercer parámetro, a pesar de ser un tipo de dato que la librería soporta por defecto (un Vector, que al mapearlo al mensaje SOAP lo transforma en un Array) los elementos que incluye el vector no son un tipo de dato que pueda ser mapeados automáticamente por la librería. Es por ello que se debe hacer uso de la clase "*PropertyInfo*" para indicar que los elementos que contiene el vector son de tipo "*NameValue*", que corresponden a un tipo complejo cuya clase encargada de realizar el mapeo se encuentra definida dentro del paquete "*sugarcrm*".

```
SoapObject request = new SoapObject(IBaseObject.NAMESPACE, "login");
request.addProperty("user_auth", userAuth);
request.addProperty("application_name", "SugarMobile");
request.addProperty(infoNameValues,nameValues);
```

A continuación se crea un objeto de tipo "*SoapObject*", que es el encargado de crear el mensaje de petición de login con los parámetros de entrada correspondientes, que posteriormente se incluye en el cuerpo del mensaje de petición SOAP. Se puede apreciar que en el constructor del objeto se le pasa tanto el espacio de nombre donde está definido el servicio web de SugarCRM, como el nombre de la función concreta a la que se desea llamar, que en este caso corresponde a la función login. Una vez creado el objeto le añadimos los parámetros de entrada de la función a través del método "*addProperty*".

```
SoapSerializationEnvelope soapSerializationEnvelope =
      createEnvelope(request);
new LoginResponse().register(soapSerializationEnvelope);
```

A continuación se crea un objeto de tipo "*SoapSerializationEnvelope*", que corresponde a una envoltura SOAP, formado por la cabecera y el cuerpo del mensaje SOAP. En el cuerpo del mensaje SOAP se incluye la información almacenada en el objeto "*request*", que es pasado por parámetro al método "*createEnvelope*". Este método realiza lo siguiente:

```
private SoapSerializationEnvelope createEnvelope(SoapObject request)
{
  SoapSerializationEnvelope soapSerializationEnvelope = new
      SoapSerializationEnvelope(SoapSerializationEnvelope.VER10);
  soapSerializationEnvelope.setOutputSoapObject(request);
  return soapSerializationEnvelope;
}
```

Crea la envoltura indicándole que la versión del mensaje SOAP será la 1.0. A continuación establece el mensaje de petición, que se encuentra incluido en el objeto "*request*" pasado por parámetro, y después devuelve la envoltura creada e inicializada.

Una vez creada la envoltura, se realiza una llamada al método "*register*" (el método comentado previamente cuando se crearon las clases para los tipos complejos) de la clase "*LoginResponse*", que es la clase encargada de indicarle a la envoltura creada las clases que debe usar para mapear los tipos de datos complejos que hay tanto en el mensaje de petición como en el de respuesta.

Una vez hecho esto sólo queda dos pasos, que son los siguientes:

```
sendRequest(soapSerializationEnvelope,"/login");
```

Realizar la llamada al método "*sendRequest*". Este método se encarga de enviar el mensaje SOAP haciendo uso del protocolo HTTP. Para ello se apoya en la clase "*HttpTrasport*", que tiene un método llamado "*call*", al que se le pasa la envoltura creada anteriormente que contiene el mensaje SOAP y una cadena indicándole la url donde se encuentra el servidor de SugarCRM al que debe enviar el mensaje. Este método crea una petición HTTP al servidor SugarCRM en cuyo cuerpo ira incrustado el mensaje SOAP.

```
EntryValue entryValue=(EntryValue) soapSerializationEnvelope.getResponse();
```

Una vez realizada la llamada, se debe recuperar los datos devueltos en el mensaje de respuesta del servidor. Eso es precisamente lo que se realiza a través del método "*getResponse*", que devuelve el tipo de dato indicado previamente en la definición de la función login, que se trata de un "*EntryValue*", conteniendo el identificador de sesión que indica que el login se realizó correctamente y que es enviado en las posteriores llamadas que se realicen al servidor para que el servidor verifique que tienes permisos para realizar esa consulta. Si por el contrario hubiera ocurrido cualquier tipo de error en el mensaje de respuesta por parte del servidor, se lanzaría la excepción "*SoapFault*", dentro de la cual se indicaría el motivo del error.

Se ha comentado la parte de código que se encarga de realizar la petición de login al servidor. Ahora se comenta la parte encargada de la recogida de los datos de entrada desde la interfaz gráfica hasta la llamada al método encargado de realizar el login de la clase ControladorSoap.

Lo primero que se muestra en la Figura 5 es la interfaz correspondiente a la ventana desde la que se realiza el login:

Figura 5. Ventana de login de usuario

Esta pantalla corresponde a la clase "*Login.java*", incluida en el paquete "com.neosistec.sugarMobile.vista". Como se puede apreciar, la interfaz está compuesta por dos campos editable correspondientes al nombre del usuario que desea realizar el login y a su password, así como de un botón desde el que se iniciaría el proceso de login. Además existe una casilla que puede ser marcada de manera que la aplicación pueda recordar al usuario que realizó el login previamente, para que no sea necesario realizarlo cada vez que la aplicación se inicia. Por lo tanto, el inicio de sesión del usuario debe ser recordado entre las distintas ejecuciones de la aplicación. Para ello se crea la clase "*ControladorConfiguracion*", cuya definición se muestra en la Figura 6:

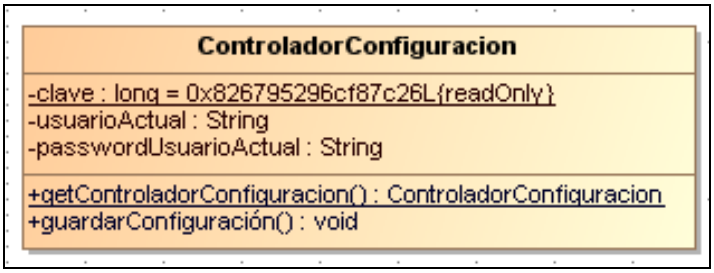

Figura 6. Diagrama de la clase "ControladorConfiguracion"

Esta clase contiene tres atributos correspondientes al usuario que realizó el inicio de sesión, a su password y a un valor correspondiente a la clave que será usada para almacenar estos valores en memoria persistente. Puede verse que esta clase tiene dos métodos públicos que son "*getControladorConfiguracion*" y "*guardarConfiguracion*". El primer método es un método estático que se implementa haciendo uso del patrón

Singleton. De esta manera, durante la ejecución de la aplicación solo existirá una instancia para este controlador que se accederá desde cualquier punto de la aplicación. Este patrón será muy usado en la aplicación, especialmente en la implementación de los distintos controladores que tendremos para la aplicación. El segundo método es el encargado de almacenar en memoria persistente la configuración que actualmente tenga. El código de este método es el siguiente:

```
public synchronized void guardarConfiguración()
{
  String[] parametros=(String[])persistentObject.getContents();

  parametros[0]=usuarioActual;
  parametros[1]=passwordUsuarioActual;
  persistentObject.commit();
}
```

Como puede apreciarse en el código, para almacenar los datos tan solo se necesita llamar al método "*getContents*" del objeto "*persistentObject*" para obtener el contenido que se encuentra almacenado en la memoria persistente (el cual corresponde a un array de String), para posteriormente modificar los datos y volver a guardarlos en memoria ejecutando el método "*commit*". La inicialización del objeto "*persistentObject*" se realiza en el constructor de la clase "*ControladorConfiguracion*":

```
/**
 *Clave para"com.neosistec.sugarMobile.controlador.ControladorConfiguracion"
 */
private static long clave=0x826795296cf87c26L;

private ControladorConfiguracion()
{
  persistentObject=PersistentStore.getPersistentObject(clave);
  String[] parametros=(String[])persistentObject.getContents();
  if(parametros==null)
  {
    // Inicializacion por defecto...
  }
  else
  {
    // Inicializacion con parametros almacenados
  }
}
```

Se realiza una primera llamada al método estático "*getPersistentObject*" pasándole como parámetro la clave que se ha creado a partir de la cadena "*com.neosistec.sugarMobile.controlador.ControladorConfiguracion*". La Figura 7. Generación de un valor de tipo String a tipo long muestra la forma de generar este valor desde Eclipse:

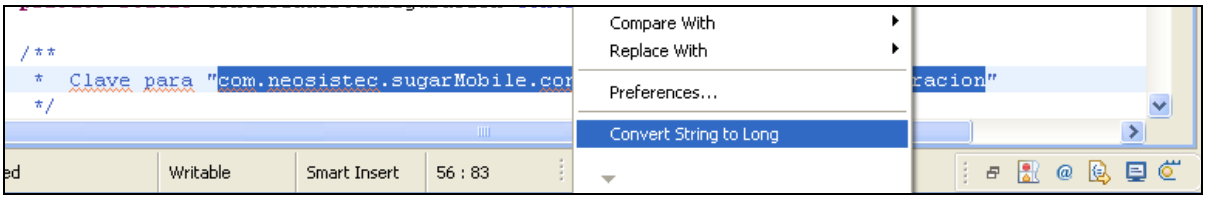

Figura 7. Generación de un valor de tipo String a tipo long

Ya ha sido creada la clase donde se almacena la sesión entre distintas ejecuciones. La siguiente clase que se crea es la del usuario. Al usar la aplicación SugarCRM se comprueba que existen varios usuarios en la aplicación que pueden estar asignados a distintas cuentas, contactos, llamadas, etc. Por lo tanto, hay que comenzar a plantearse cuales van a ser los "*beans*" que formarán parte de nuestra aplicación. Dado que en este momento sólo interesa realizar el login con el servidor, se simplifica la clase "*Usuario*" incluyéndole tan solo los atributos de nombre de usuario e identificador. Esta clase esta dentro del paquete "*com.neosistec.sugarMobile.modelo.beans*", que contendrá todos los beans usados en la aplicación como las cuentas, contactos, llamadas, tareas, usuarios, etc. A continuación, se implementa el controlador de la clase "*Usuario*", que se encargue de enlazar la interfaz con el resto de componentes de la aplicación. La definición de la clase "*ControladorUsuario*" y "*Usuario*" se muestra en la Figura 8:

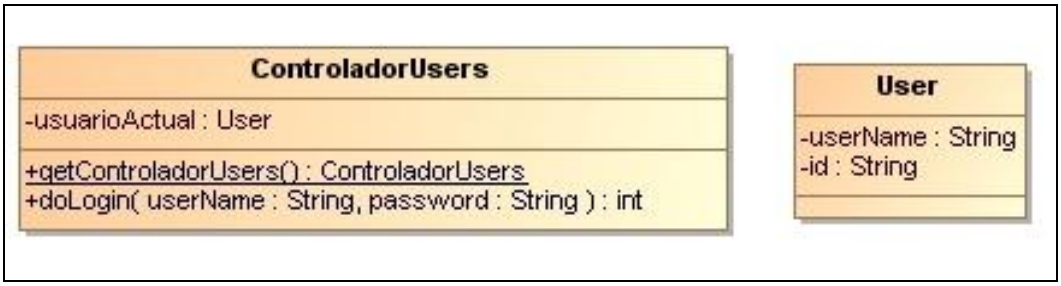

Figura 8. Diagrama de clases de ControladorUser y User

Una vez que se introduzcan los valores de usuario y password dentro de los campos correspondientes y se pulse el botón enviar se realiza una llamada al método "*doLogin*" de la clase "*ControladorUsers*", que es la encargada de llamar al método "*doLogin*" de la clase "*ControladorSoap*", que finalmente se comunica con el servidor. Un aspecto importante a tener en cuenta es que cuando se pulse el botón enviar, el código que se ejecuta no llama directamente al método "*doLogin*" de la clase "*ControladorUsers*", ya que se debe recordar que el hilo que ejecuta este código corresponde al hilo despachador de eventos, que no puede realizar operaciones que consuman mucho tiempo (como por ejemplo una consulta a través de la red), ya que no se podría atender a los eventos de entrada que se produjeran durante ese tiempo. Para evitarlo, lo que se hace es crear un nuevo hilo que se encarga de realizar el login. Cuando se realice el login de forma correcta, se comprueba si estaba marcada la casilla de guardar sesión, actualizando los valores de "*ControladorConfiguracion*" en caso afirmativo. La Figura 9

muestra un diagrama de secuencia donde se puede ver la interacción entre las clases que realizan el login del usuario:

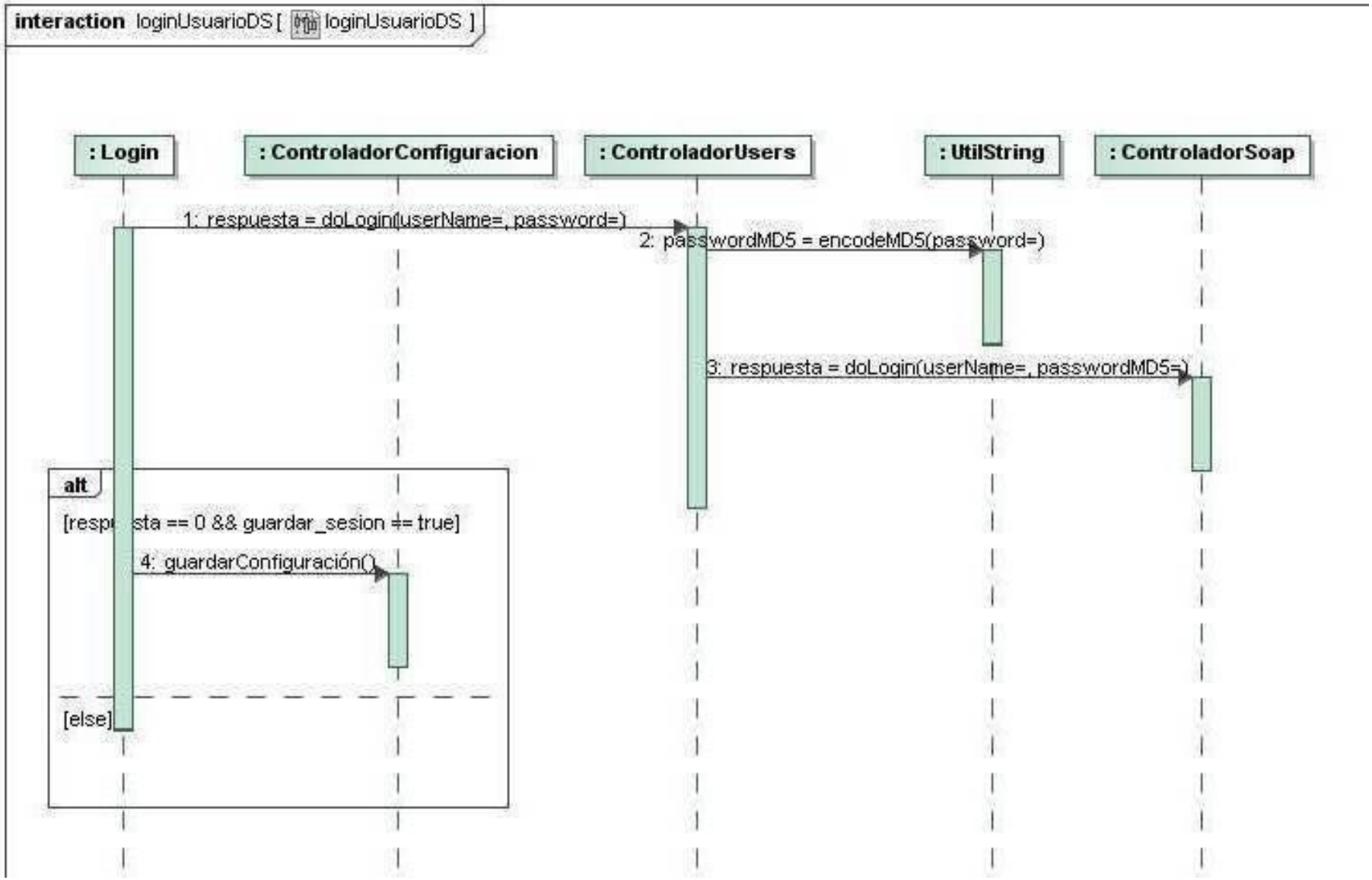

Figura 9. Diagrana de secuencia login de usuario

Aquí termina el desarrollo de esta iteración, con el prototipo funcional de la aplicación funcionando y realizando correctamente el login con el servidor. Para acabar se planifica la funcionalidad de la aplicación que desarrollaremos para la siguiente iteración. Si se comprueba la aplicación web de SugarCRM, lo primero que ve tras realizar el login es una página principal donde se muestran algunos aspectos relevantes para el usuario, como las actividades que tiene que hacer (llamadas, reuniones, tareas), las oportunidades abiertas que tiene, etc. Además puede verse como en la parte superior tiene un menú desde el cual puede acceder a los distintos módulos de la aplicación como las cuentas, contactos, actividades, etc., como puede verse en la Figura 10:

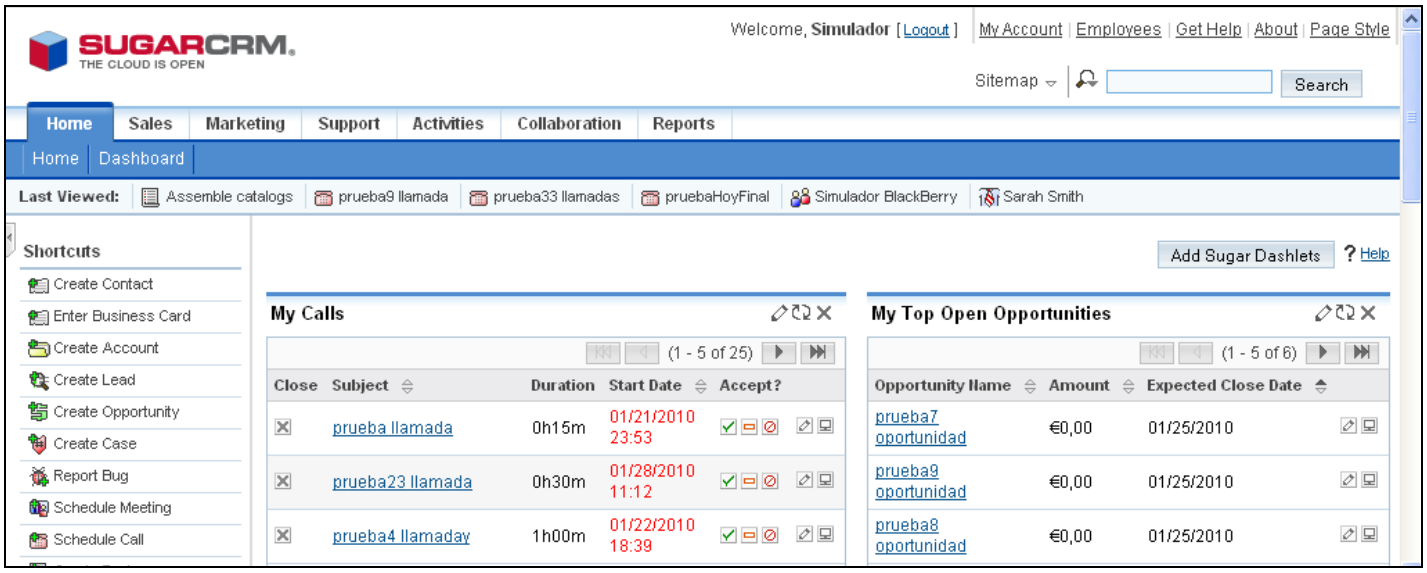

Figura 10. Página principal SugarCRM

Como generalmente los usuarios de la aplicación estarán acostumbrados a usar la aplicación web de SugarCRM, la interfaz gráfica se diseña de manera semejante a la aplicación web, principalmente en lo que a la navegabilidad y número de ventanas para cada módulo que puedan encontrar. Se podría plantear para la siguiente iteración implementar esta pantalla principal desde la que el usuario pueda recorrer los distintos módulos de la aplicación, así como la de tener un acceso rápido a los aspectos más relevantes del usuario para cada módulo. El problema es que se deberían implementar todos los módulos de SugarCRM accesibles desde la aplicación, por lo que el diseño de esta ventana principal se dejará para más adelante. En la siguiente iteración se implementa un módulo, concretamente el módulo de cuentas, mostrado en la Figura 11:

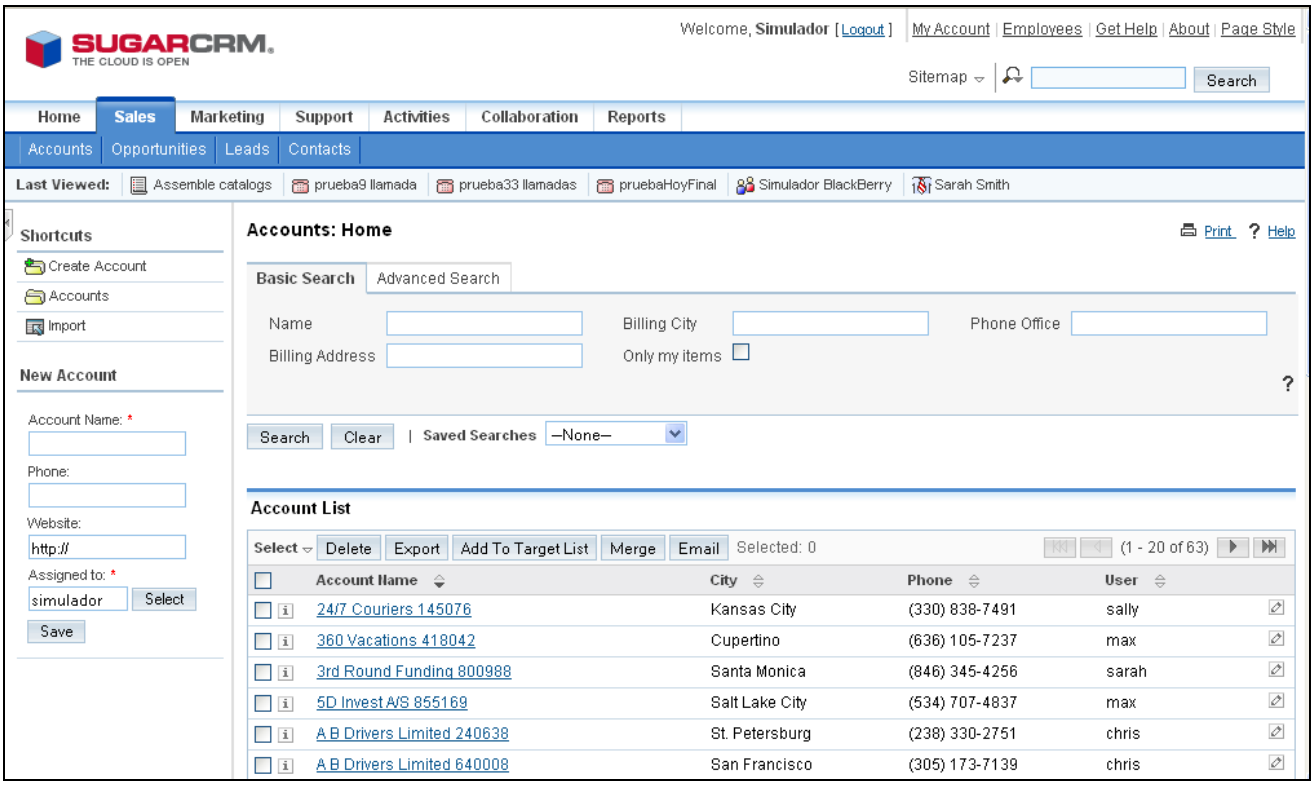

Figura 11. Página principal del módulo Cuentas

Como puede verse, la página principal del módulo cuentas muestra una serie de opciones: crear cuentas, ver un listado de cuentas o hacer una búsqueda filtrada de las cuentas. Para la siguiente interacción se realiza la implementación necesaria para que nuestra aplicación pueda mostrar un listado de cuentas.

3.2.3 3ª Iteración (Duración - 2 semanas):

Para esta iteración se desarrolla la funcionalidad necesaria para que puedan verse en el dispositivo móvil un listado de las cuentas existentes. En este punto se empieza a trabajar con un punto fuerte de aplicación, que son los módulos de la aplicación SugarCRM (Cuentas, Usuarios, Contactos, Llamadas, Reuniones, Oportunidades, etc.) y que hay que diseñar correctamente para trabajar con toda esta cantidad de información. De aquí surgen dos preguntas: "¿Qué atributos o campos existen en cada módulo?" Y "¿Cómo están relacionados los módulos entre sí?". Para responder a estas preguntas se usan dos funciones de los servicios web ofrecidos por SugarCRM. La primera de ella corresponde a la función "*get_available_modules*" cuya definición se muestra en la Figura 12:

Figura 12. Definición de la función "get_available_modules"

Como puede verse en la imagen, esta función recibe como parámetro el identificador de sesión obtenido cuando el usuario realiza el login, y devuelve una lista con todos los módulos disponibles para este usuario. La segunda función es la llamada "*get_module_fields*", cuya definición se muestra en la Figura 13:

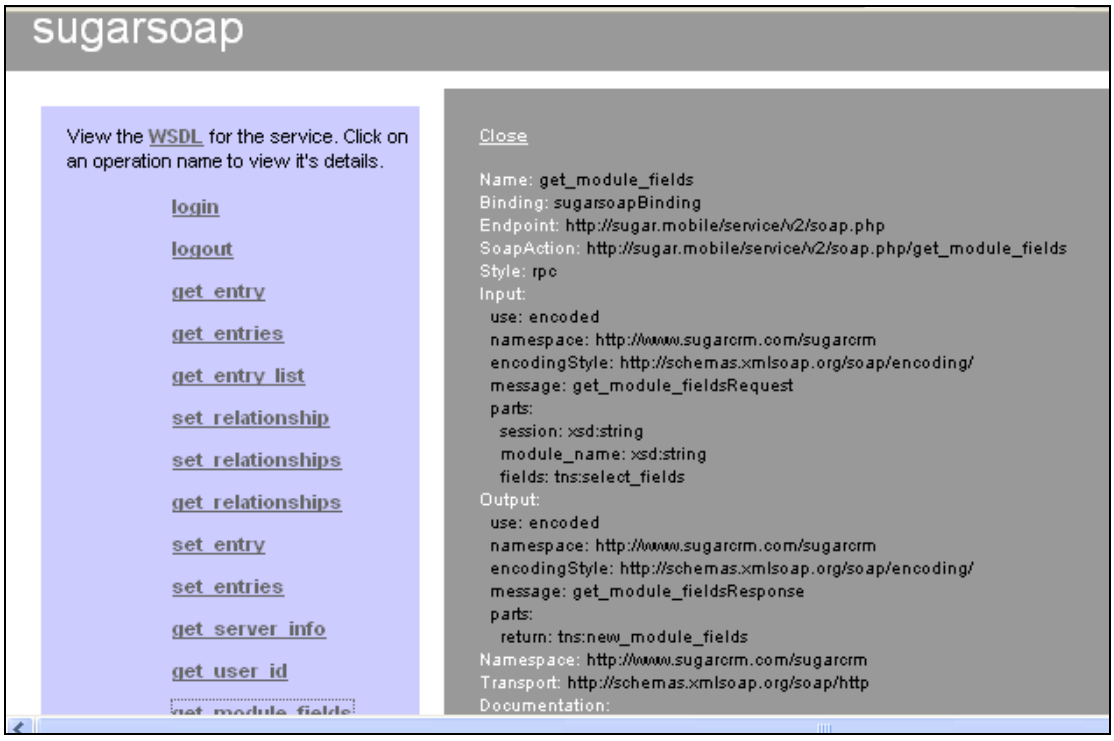

Figura 13. Definición de la función "get_module_fields"

Como puede verse, la función recibe tres parámetros de entrada correspondientes a:

- *session*: La sesión de usuario que recibió al realizar el login.
- *module_name*: El nombre del módulo del que se quiere obtener la información.
- *fields*: Lista de los parámetros que se desea obtener. Es opcional, si se deja vacío se obtendrán todos los parámetros del módulo.

Esta función devolverá un vector con dos valores: por un lado una lista con los campos que forman parte de este módulo, y por otro lado una lista con las relaciones existentes entre este módulo y los demás. Al ejecutar ambas funciones podemos obtener un listado de todos los módulos con sus respectivos campos y relaciones, y guardarlos en un archivo para poder comenzar la implementación de estos módulos para nuestra aplicación.

De estas dos funciones obtenemos los campos necesarios para los registros del módulo Cuenta, lo siguiente que se debe plantear es cómo realizar la implementación del listado de cuentas en el dispositivo BlackBerry. Una opción sería hacerlo igual que la aplicación web de SugarCRM, realizando una consulta al servidor cada vez que se desee mostrar el listado de cuentas, pero eso no sería factible, ya que incumpliría con tres de los objetivos que deseamos conseguir para esta aplicación, que son: bajo uso de la red, bajo consumo de batería y uso de modo offline. Por lo tanto se llega a un punto clave de la aplicación, y es que la información que se descargue de los distintos módulos será almacenada en el dispositivo para poder ser consultados posteriormente sin necesidad de realizar otra consulta al servidor. Previamente comentamos que existen varias formas de almacenar datos en la memoria persistente del dispositivo, y que de todas ellas se usa la de BlackBerry Persistent Store. Pero lo ideal sería abstraer la aplicación del modo en que se almacena los distintos beans usados en nuestra aplicación, por lo que se hará uso del patrón DAO para resolver este problema. De esta forma la aplicación trabajará con los beans de Cuenta, Contacto, etc., sin preocuparse si esa información es almacenada en la memoria flash de la BlackBerry, en una tarjeta SD o en cualquier otro dispositivo de almacenamiento. La Figura 14 muestra el diagrama de clases del patrón DAO para el módulo Cuenta de la aplicación:

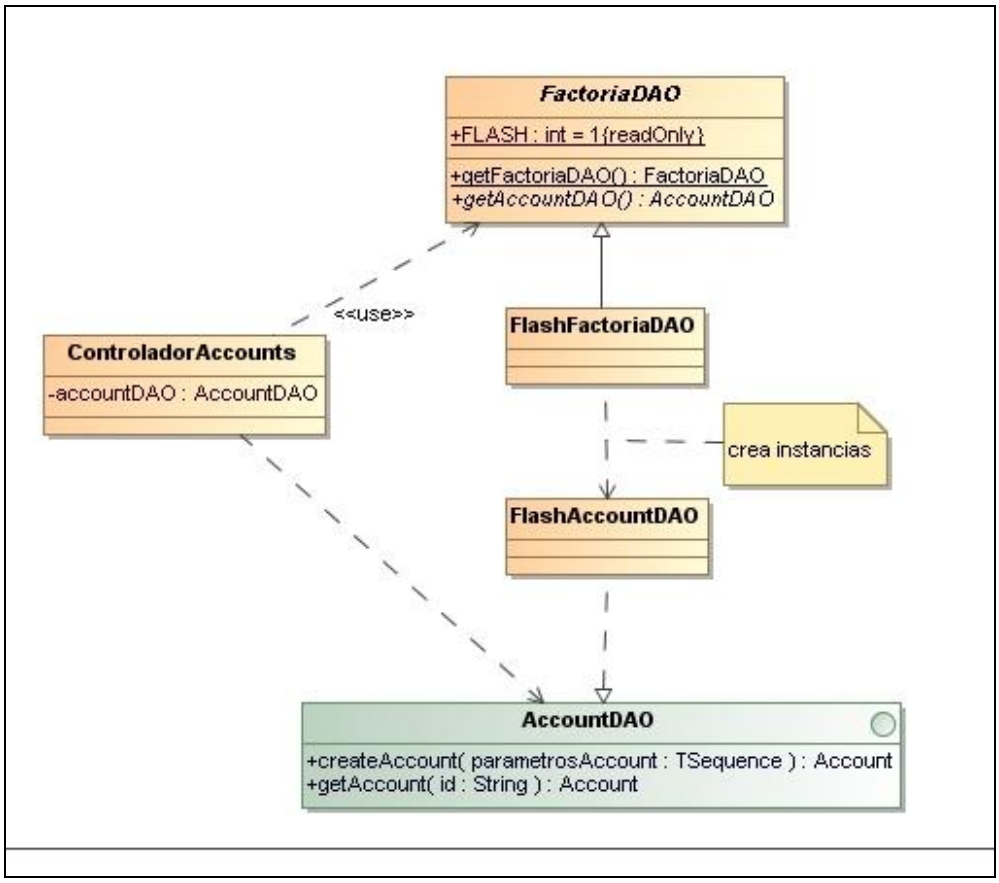

Figura 14. Patrón DAO aplicado a Cuentas

Los siguientes puntos a resolver son: descargar las cuentas del servidor al dispositivo y mostrar en pantalla el listado de cuentas.

Para realizar la descarga se toma la decisión de que el usuario pueda realizar un búsqueda en el servidor indicando que cuentas desea descargar, aplicando unas opciones de filtrado. La Figura 15 muestra el diseño de la pantalla de búsqueda de cuentas:

Figura 15. Ventana para descargar cuentas del servidor

Donde lo que se intenta es simular la búsqueda básica que se puede encontrar en la aplicación web de SugarCRM cuando se muestra el listado de cuentas. Para realizar la búsqueda en el servidor se hace uso de la función "*get_entry_list*", mostrada en la Figura 16:

Figura 16. Definición de la función "get_entry_list"

Esta función se encarga de devolver los registros del módulo indicado y que cumplan con los parámetros de filtrado que se le pasan. Los parámetros de la función son los siguientes:

- *session*: Identificador de sesión de usuario que recibió con el login.
- *module_name*: El nombre del módulo donde obtener los registros.
- *query*: Corresponde al filtro añadido a la consulta. Irá dentro de la clausula "*where*" de SQL.
- *order_by*: Lo mismo que el anterior pero para la clausula "*order by*" de SQL.
- *offset*: Número de registros que se deben descartar en la respuesta de la consulta realizada.
- *select_field*: Lista de los campos del registro a descargar. Es opcional.
- *link_name_to_fields_array*: Este parámetro nos permitiría obtener valores de otros registros pertenecientes a otros módulos relacionados con el registro de la consulta actual.

- *max_results*: El número máximo de registros a devolver.
- *deleted*: Indica si los registros que han sido borrados deben ser incluidos en la respuesta o no.

A continuación se define el tipo complejo "*get_entry_list_result_version2*" devuelto por la función:

```
- <xsd:complexType name="get_entry_list_result_version2">
  - <xsd:all>
      <xsd:element name="result_count" type="xsd:int"/>
      <xsd:element name="next_offset" type="xsd:int"/>
      <xsd:element name="entry_list" type="tns:entry_list"/>
      <xsd:element name="relationship_list" type="tns:link_lists"/>
    </xsd:all>
  </xsd:complexType>
```

Donde:

- *result_count*: Número de registros devueltos.
- *next_offset*: El inicio del siguiente registro no devuelto y que cumple con la consulta.
- *entry_list*: Lista con los registros que fueron devueltos.
- *relationship_list*: Lista con los registros relacionados, si el parámetro "*link_name_to_fields_array*" fue especificado.

La Figura 17 muestra el diagrama de secuencia correspondiente a la descarga de registros desde el servidor al dispositivo:

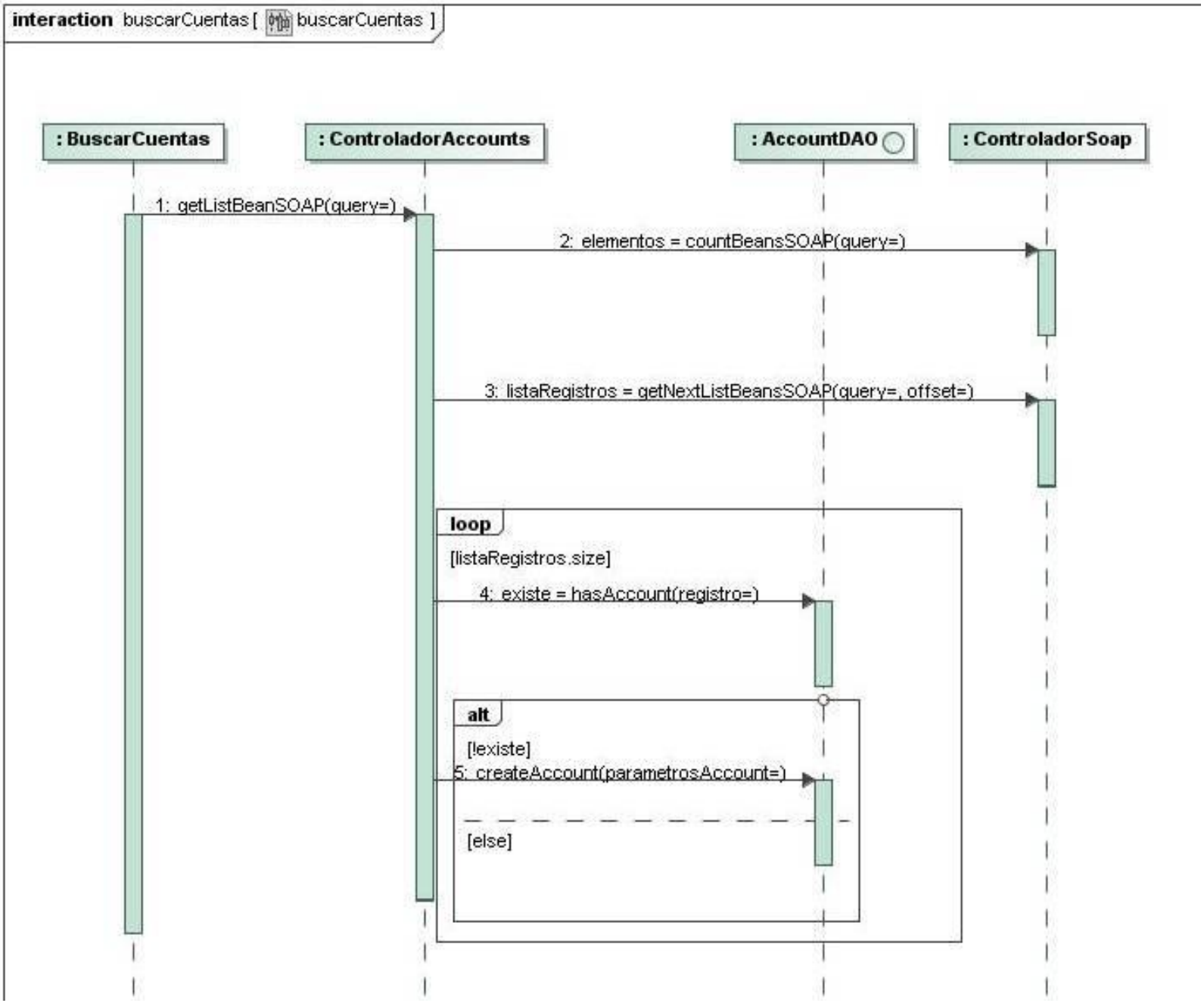

Figura 17. Diagrama secuencia de búsqueda de cuentas

Como se puede ver en el diagrama anterior, existe una llamada al método "*ControladorSoap.countBeansSoap(String query)*" que devuelve el número de registros existentes en un módulo especificado. La función usada del servicio web de SugarCRM para realizar esta acción es "*get_entries_count*", mostrada en la Figura 18:

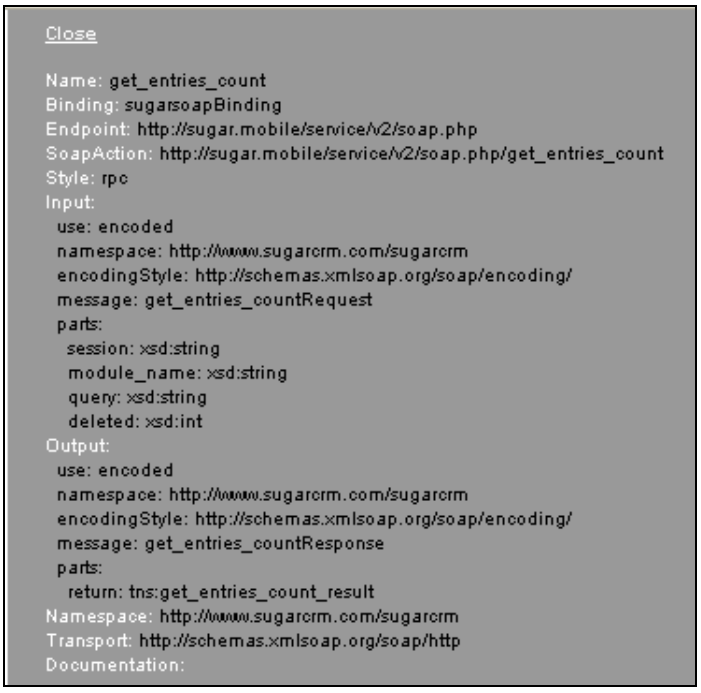

Figura 18. Definición de la función "get_entries_count"

Los parámetros de la función son los siguientes:

- *session*: Identificador de sesión de usuario que recibió con el login.
- *module_name*: El nombre del módulo donde obtener los registros.
- *query*: Corresponde al filtro añadido a la consulta. Va dentro de la clausula "where" de SQL.
- *deleted*: Indica si los registros que han sido borrados deben ser incluidos en la respuesta o no.

La función devuelve el tipo complejo "*get_entries_count_result*" definido como:

```
– <xsd:complexType name="get_entries_count_result">
  – <xsd:all>
      <xsd:element name="result_count" type="xsd:int"/>
    </xsd:all>
  </xsd:complexType>
```

Donde:

- *result_count*: Número de registros existentes en el módulo y que cumplan con la consulta.

Esta función se usa para conocer rápidamente cuantos registros se descargan con esa consulta, y de esta manera poder indicarle al usuario de la aplicación en tiempo real cuantos registros se han descargado y cuantos están aún por descargar.

Se ha implementado la lógica necesaria para realizar la descarga de cuentas desde el servidor al dispositivo, quedando por realizar el listado de las cuentas en el dispositivo. A la hora de realizarlo, debido a que el número de cuentas que podría llegar a haber en el dispositivo podría ser elevado, se considera que la forma más legible de mostrarle los listados al usuario sería de forma paginada, mostrando por pantalla un número máximo de cuentas del total que existían almacenadas en el dispositivo. Además se considera añadirle un pequeño buscador, de manera que el usuario pueda rápidamente acceder a la cuenta que desea. La ordenación de las cuentas se realizó por el nombre de las mismas. Para ello, se hace uso de una estructura de almacenamiento auxiliar, concretamente de un vector ordenado. La clase concreta usada que se encarga de realizar esta función es la clase "*SimpleSortingVector*", y se le indica que debe ordenar sus elementos ejecutando las siguientes instrucciones:

```
simpleSortingVectorAccounts=new SimpleSortingVector();
simpleSortingVectorAccounts.setSort(true);
simpleSortingVectorAccounts.setSortComparator(new SortAccountsByName());
```

Con la segunda instrucción se le indica que debe comenzar a ordenar los elementos que se le vayan introduciendo, y con la tercera instrucción se le indica de qué forma debe ordenar esos elementos. Esto se consigue pasándole una clase que implemente la interfaz "*Comparator*", que en nuestro caso es la clase "*SortAccountByName*", que se encarga de comparar las cuentas según su nombre. Si se desea en cualquier otro momento ordenar las cuentas por otro campo, como por ejemplo la fecha de creación, tan solo se debe crear otra clase que implemente la interfaz "*Comparator*" y que realice la comparación por fecha de creación de las cuentas, y asignársela al vector ordenado a través del método "*setSortComparator(new SortAccountsByDate)*".

Para controlar la paginación en el listado se crea un atributo en la clase "*ControladorAccount*" que indica el "*offset*" actual que lleva el listado de cuentas, así como contendrá un vector en el que se encontrarán las cuentas que actualmente cumplan con el filtrado añadido por el usuario, si es que existiera. En la Figura 19 se muestra un diagrama de secuencia correspondiente al listado de cuentas:

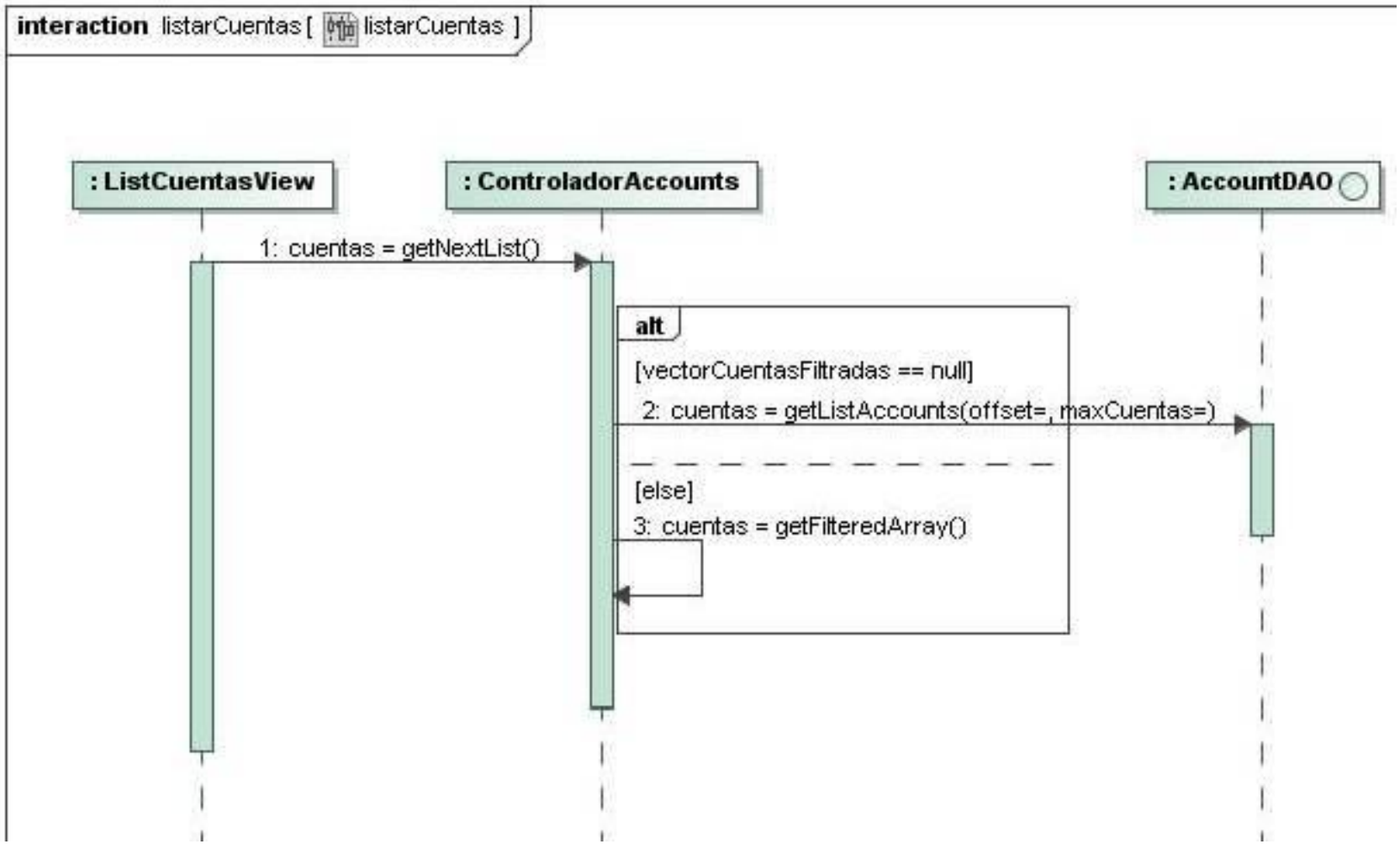

Figura 19. Diagrama de secuencia listado de cuentas

El listado de cuentas muestra un poco de información sobre cada cuenta. Para acceder a la información completa puede seleccionar un elemento de la lista y aparecerá la ventana de detalles, la cual tiene una característica especial que será comentada en la iteración correspondiente a las relaciones entre registros. La Figura 20 muestra la ventana de detalles de una cuenta:

Figura 20. Ventana de detalles de una cuenta

Una vez finalizada la implementación se obtiene un nuevo prototipo de la aplicación que puede mostrar un listado de cuentas, filtrar las mismas, ver los detalles de una cuenta y realizar una búsqueda de cuentas en el servidor, como se muestra en la Figura 21:

Figura 21. Ventana con el listado de cuentas descargadas desde el servidor

Llegado a este punto, se debe comentar un "error" crítico que se pudo apreciar en la aplicación. Si se revisa el fichero obtenido por la función "*get_module_fields*" para el módulo Cuenta, el resultado obtenido es el siguiente:

```
++++++Modulo ==> Accounts

FIELDS:

        * name => id
        * type => id
        * label => ID
        * required => 0
        * default value => null
        * options =>

        * name => name
        * type => name
        * label => Name:
        * required => 1
        * default value => null
        * options =>

        * name => date_entered
        * type => datetime
        * label => Date Entered:
        * required => 0
        * default value => null
        * options =>

        .
        .
        .
```

Para simplificar sólo se muestra parte del resultado obtenido. Como se puede apreciar, un registro perteneciente al módulo "*Accounts*" (Cuenta) estará formado por un campo correspondiente al "*id*" de la cuenta, otro campo correspondiente al "*name*" de la cuenta, y así sucesivamente con los campos que se consideren relevantes para la aplicación. Esto significa que la clase Account (que será un objeto) se define con los atributos "*id*" y "*name*" que serán de tipo String (otros objetos), y de la misma forma con el resto de campos. A día de hoy, el SO BlackBerry define dos tipos de manejadores de objetos: Transitivos y Persistentes.

Los manejadores de objetos transitivos, comúnmente conocidos como "manejadores de objetos", son consumidos por cada objeto existente en el sistema.

Los manejadores de objetos persistentes, por otro lado, sólo son consumidos cuando un objeto es persistente. Es decir, un objeto persistente consume tanto un manejador transitivo como uno persistente.

Existen un número fijo de manejadores en el sistema, dónde generalmente el número de manejadores persistentes es la mitad al número de

manejadores transitivos, y este número depende del tamaño de la memoria del dispositivo. En la Figura 22 se muestran algunos ejemplos:

Flash Memory	Persistent Object Handles	Object Handles
8 MB	12,000	24,000
16 MB	27,000	56,000
32 MB	65,000	132,000

Figura 22. Manejadores de objetos según tamaño de memoria

Por lo tanto se debe ser muy cuidadoso en el uso de los manejadores persistentes, ya que son un bien escaso. En la aplicación, cuando se almacena una cuenta, si sólo se tiene en cuenta los campos mostrados previamente, se está usando un manejador persistente correspondiente a la cuenta, más otros tres correspondientes al identificador de la cuenta, al nombre de la cuenta y a la fecha de creación. Esto significa que para cada cuenta se están usando 4 manejadores persistentes. Si se guardan 100 cuentas, se están usando 400 manejadores persistentes, y eso sólo para almacenar cuentas, por lo que el número incrementará cuando se guarde contactos, llamadas, reuniones, oportunidades, etc., cada una con sus correspondientes campos, cuya media es de unos 10 campos por registro. Como se puede apreciar, si bien no se puede considerar un error de implementación (debido a que si "BlackBerry Persistent Store" trabaja así no se puede hacer nada al respecto), si se puede considerar un error de planteamiento en la viabilidad del proyecto, ya que puede apreciarse fácilmente que la aplicación hace un uso abusivo de los manejadores de objetos persistentes. Además se debe tener en cuenta que los manejadores se comparten por todas las aplicaciones que se ejecuten en el dispositivo, lo que agrava más aún el problema.

Afortunadamente, los ingenieros de RIM solucionaron este problema creando una nueva clase incluida a partir de la versión 4.0 de la API BlackBerry llamada "*ObjectGroup*" que, como su propio nombre indica, tiene la capacidad de agrupar objetos bajo un único manejador de objetos. Volviendo al caso anterior, al almacenar una cuenta usando esta clase la cuenta solo consumiría un manejador persistente (frente a los 4 de antes), mientras que si se almacenan 100 cuentas se consumirían 100 manejadores persistentes (frente a los 400 de antes). Como puede apreciarse, el uso de la clase ObjectGroup hace que sea viable el uso de la aplicación en una BlackBerry, al disminuir considerablemente el uso de manejadores persistentes. Sin embargo, existen dos inconvenientes al uso de esta clase, que son:

1) Cuando un objeto es agrupado es considerado sólo de lectura. Esto no significa que un objeto agrupado no pueda ser modificado, sino

que para modificarlo primero hay que desagruparlo, realizar las modificaciones oportunas y posteriormente volverlo a agrupar. Si se intenta modificar un objeto agrupado el SO lanza la excepción "*ObjectGroupReadOnlyException*".

2) Existe una penalización en el rendimiento cuando un objeto es desagrupado. El sistema realizará una copia del objeto agrupado y reasignara manejadores a cada uno de los objetos que estén dentro del grupo. Por lo tanto, esta operación solo debería realizarse cuando sea estrictamente necesario.

Aquí termina esta iteración con el prototipo funcional de la aplicación funcionando, y habiendo solucionado el problema surgido con la persistencia de los registros en el dispositivo. Llegado a este punto se considera que ya se han abordado los aspectos más críticos para el desarrollo de la aplicación, por lo que se considera terminada esta fase de elaboración y se continúa con la fase de construcción para completar el desarrollo de la aplicación. En la primera iteración de esta fase se realiza la implementación necesaria para que la aplicación sea capaz de crear nuevas cuentas y de modificar las cuentas descargadas desde el servidor, y que el servidor sea avisado de tales sucesos para que actúe en consecuencia.

3.3 Fase de Construcción

3.3.1 1ª Iteración (Duración – 4 días):

En esta iteración se realizan la codificación necesaria para crear y actualizar cuentas. Para realizar una modificación de una cuenta se deben realizar dos pasos: almacenar los nuevos valores de la cuenta en la memoria persistente y enviarlos al servidor para que se actualice.

La función que nos ofrecen los servicios web de SugarCRM para realizar la actualización de la cuenta es "*set_entry*", definida en la Figura 23:

Figura 23. Definición de la función "set_entry"

Los parámetros de entrada de la función son:

- *session*: Identificador de sesión obtenida al realizar el login.
- *module_name*: El nombre del módulo al que pertenece el registro a modificar.
- *name_value_list*: Lista de pares <campo, valor> correspondientes a los campos del registro. Este array está definido de la siguiente forma:

```
- <xsd:complexType name="name_value">
  - <xsd:all>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
- <xsd:complexType name="name_value_list">
  - <xsd:complexContent>
    - <xsd:restriction base="SOAP-ENC:Array">
        <xsd:attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="tns:name_value[]"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
```

El parámetro "*name_value_list*", al tratarse de una actualización de un registro, debe llevar obligatoriamente el valor del campo correspondiente al identificador del registro. SugarCRM identifica cada registro perteneciente a cualquier módulo a través de un identificador único que le es asignado al

crearse, generado a partir de una función que devuelve como valor un identificador de 36 caracteres únicos. Este identificador también se usa para saber qué registros de un módulo están relacionados con otros registros del mismo módulo o de otros. Si no se añade el identificador, SugarCRM supondrá que se está creando un nuevo registro y lo creará asignándole su nuevo identificador y el resto de valores que se le pase en la variable "*name_value_list*".

El valor devuelto por la llamada corresponde al tipo complejo "*new_set_entry_result*", definido cómo:

```
− <xsd:complexType name="new_set_entry_result">
  − <xsd:all>
      <xsd:element name="id" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
```

Este tipo está formado por un solo atributo, que corresponde al identificador del registro que ha sido actualizado (o creado, según el caso), o un valor de "-1" si se produjo algún error. En la Figura 24 se muestra el diagrama de secuencia correspondiente a la actualización de una cuenta:

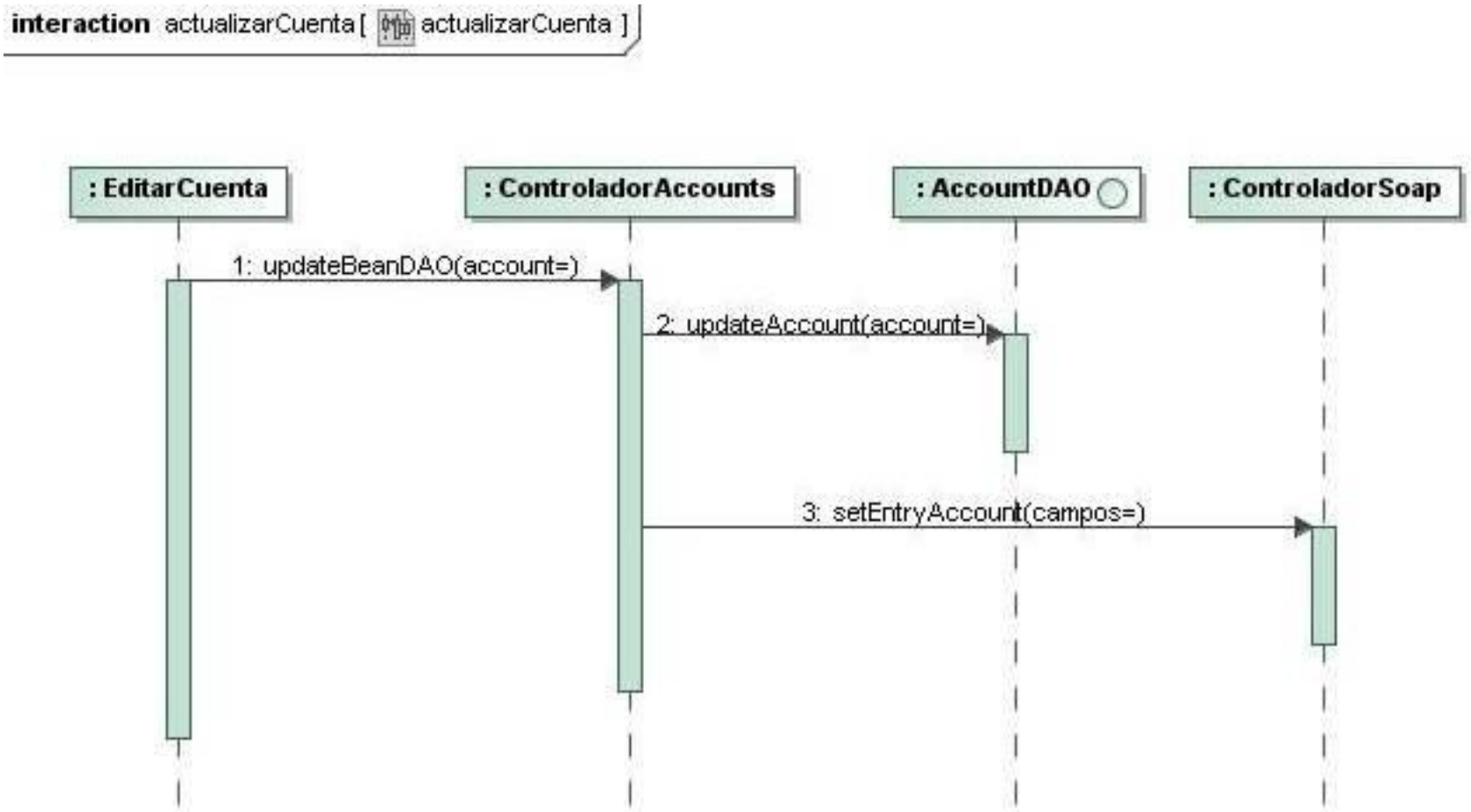

Figura 24. Diagrama de secuencia edición cuenta

Para la creación de una nueva cuenta, la función usada de los servicios web de SugarCRM será la misma que para la actualización, pero en este caso no se le pasa el valor del identificador de la cuenta, por lo que el servidor interpreta que se trata de un registro nuevo y lo crea, devolviéndonos el nuevo identificador de la cuenta. Aquí surge un problema, y es que uno de los objetivos que se desea conseguir es que la aplicación pueda funcionar en modo offline, sin embargo, no se puede trabajar con un nuevo registro de cualquier módulo (modificarlo, asignarle relaciones o eliminarlo) si ese registro no tiene un identificador único. Para solucionar este problema, cuando se crea un nuevo registro, la aplicación le asigna un identificador temporal único, de manera que ese registro puede ser usado de la misma forma que otro registro que tenga un identificador asignado por el servidor. Cuando se realice la llamada al servidor "*set_entry*" para crear ese nuevo registro y nos devuelva el valor del identificador asignado por el servidor, el identificador temporal del registro será substituido por el que le ha asignado el servidor, y se actualizarán las relaciones de este registro con el resto de registros de la aplicación para que hagan referencia al valor correcto del identificador asignado por el servidor. Para acabar, la Figura 25 muestra el diagrama de secuencia correspondiente a la creación de una cuenta:

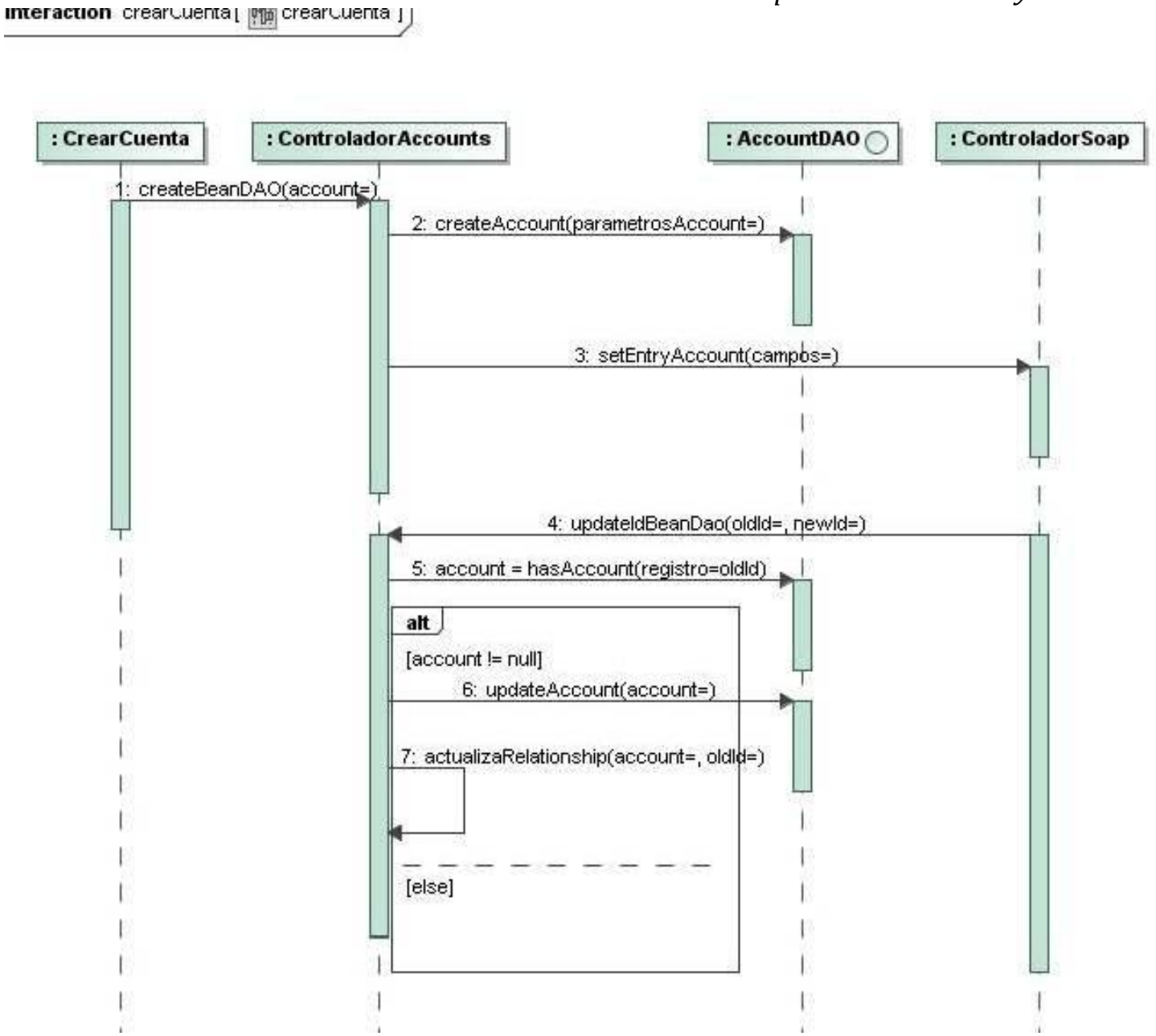

Figura 25. Diagrama de secuencia crear cuenta

El método "*actualizaRelationship*" de la clase ControladorAccounts se comenta en la siguiente iteración dedicada a las relaciones entre registros.

Aquí termina esta iteración con el prototipo funcional de la aplicación funcionando. En la Figura 26 pueden verse las ventanas correspondientes a la creación y actualización de una cuenta:

Figura 26. Ventanas de creación y edición de una cuenta

En la siguiente iteración se realiza la implementación de las relaciones existentes entre los registros de un módulo, o con los de otro módulo.

3.3.2 2ª Iteración (Duración – 4 días):

Todos los módulos existentes en SugarCRM se encuentran relacionados entre sí. En esta iteración realizaremos la implementación necesaria para que pueda relacionarse una cuenta con los módulos a los que pueda relacionarse, así como poder descargar del servidor las relaciones que previamente se establecieran. Estas relaciones aparecen en la aplicación web SugarCRM cuando se muestran los detalles de una cuenta. La Figura 27 corresponde a un diagrama ER (entidad-relación) dónde se puede ver las relaciones del módulo cuenta con el resto de módulos:

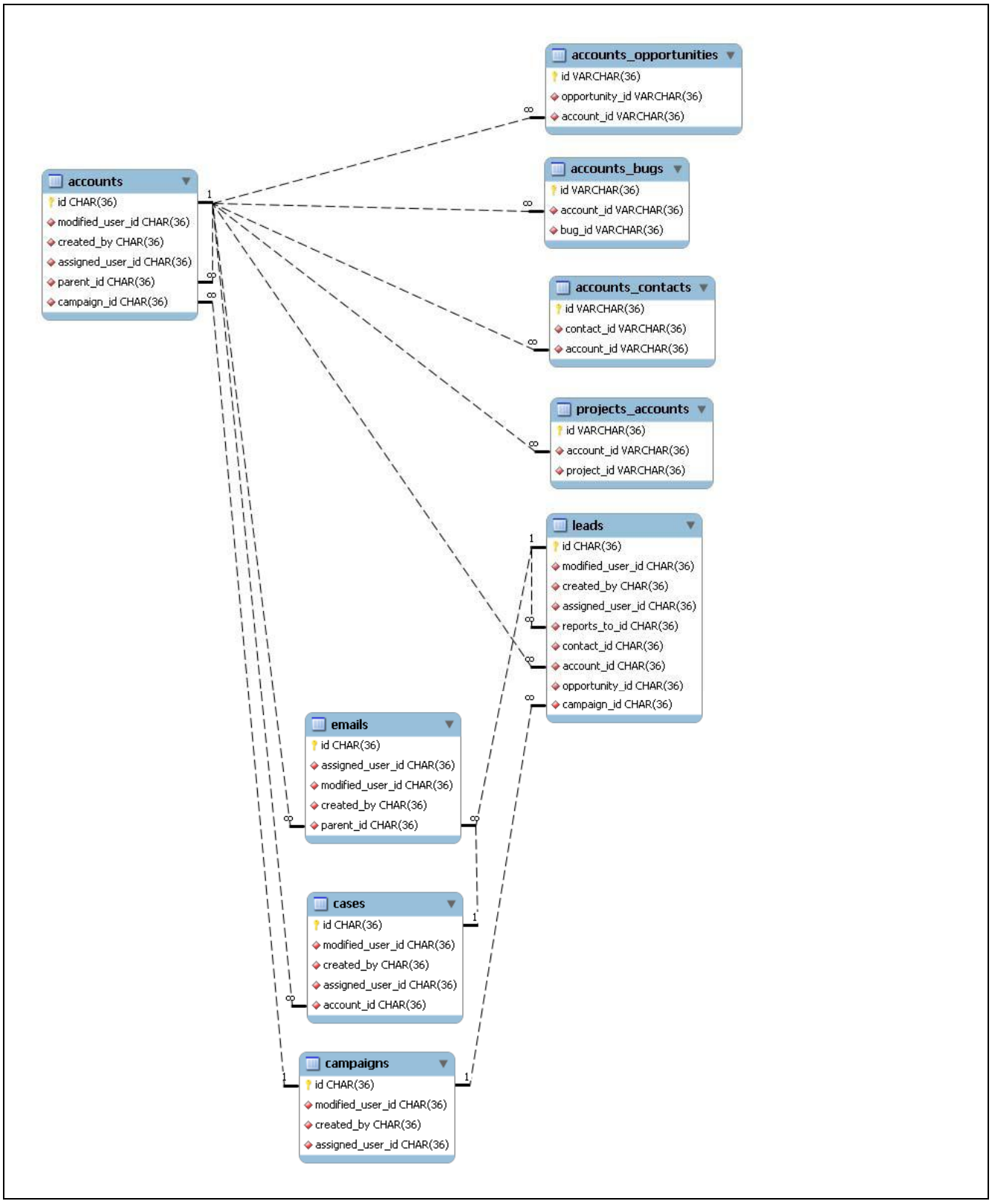

Figura 27. Diagrama ER del módulo Cuentas

Como se puede observar, el módulo Cuenta se relaciona consigo mismo y con otros módulos como Contactos, Oportunidades, Casos, Actividades, etc. Esto significa que en este punto se tienen que crear los beans que representen los registros de todos de módulos con los que se relaciona una

cuenta, aunque no se van a implementar completamente, tan solo se les añade el único campo necesario para realizar las relaciones: el identificador del registro.

Como se puede apreciar en el diagrama ER, una cuenta tiene tanto relaciones de tipo uno-a-uno como relaciones de tipo uno-a-muchos y muchos-a-muchos. Por lo tanto, las relaciones de uno-a-uno se implementan añadiendo un atributo a la clase Cuenta correspondiente al identificador de la otra clase (Oportunidad, Contacto, etc.) con la que está relacionada. Para las relaciones uno-a-muchos y muchos-a-muchos, se añade una lista para cada módulo con el que se relacione dónde una la cuenta va añadiendo los identificadores de los registros con los que se relaciona.

La función encargada de establecer una relación entre dos registros es "*set_relationship*". La Figura 28 muestra la definición de la misma:

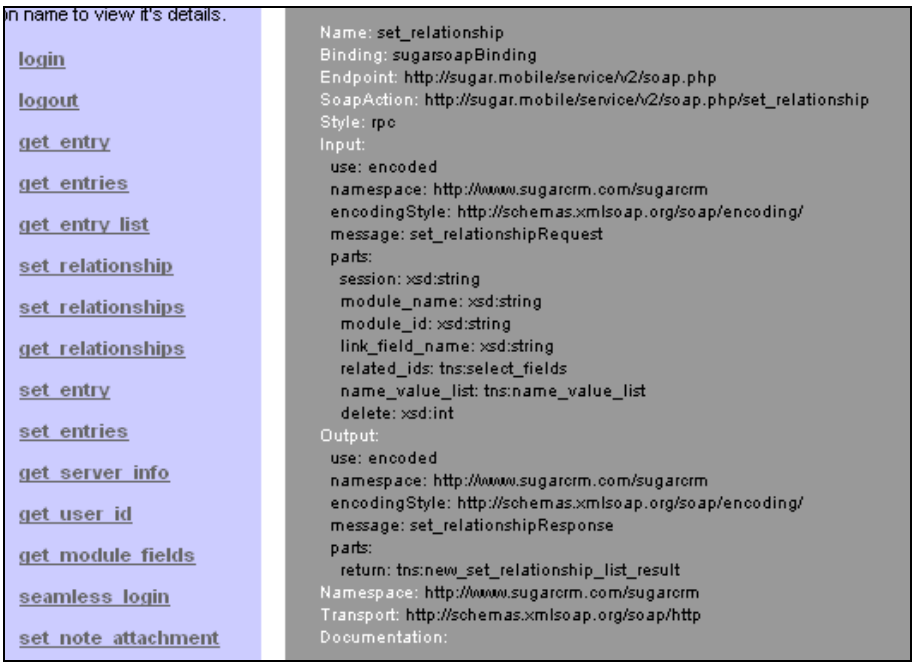

Figura 28. Definición de la función "set_relationship"

Los parámetros de entrada son:

- *session*: Identificador de sesión obtenida al realizar el login.
- *module_name*: Nombre del módulo del registro que va a realizar la relación.
- *module_id*: El identificador del registro del módulo especificado en module_name.
- *link_field_name*: Nombre del tipo de relación que se desea realizar.

- *related_ids*: Lista de los identificadores con los que se desea relacionar.

- *name_value_list*: Lista de pares <clave,valor> correspondiente a campos del registro que pueden ser modificados. Es opcional.

- *delete*: Si el valor es 0 la relación es establecida, si es 1 la relación es borrada.

El parámetro "link_field_name" debe tener uno de los valores obtenidos al ejecutar la función "*get_module_fields*" para el módulo Cuentas. Al revisar el fichero se obtiene la siguiente información:

```
++++++Modulo ==> Accounts
                .
                .
                .
LINK_FIELDS:
        * name => assigned_user_link
        * type => link
        * relationship => accounts_assigned_user
        * module => Users
        * bean name => User

        * name => member_of
        * type => link
        * relationship => member_accounts
        * module => Accounts
        * bean name => Account

        * name => cases
        * type => link
        * relationship => account_cases
        * module => Cases
        * bean name => aCase

        * name => tasks
        * type => link
        * relationship => account_tasks
        * module => Tasks
        * bean name => Task

        * name => meetings
        * type => link
        * relationship => account_meetings
        * module => Meetings
        * bean name => Meeting

        * name => calls
        * type => link
        * relationship => account_calls
        * module => Calls
        * bean name => Call
        .
        .
        .
```

Concretamente, el valor que se pone corresponde al asignado en el campo "*relationship*". Cada módulo tendrá sus propios nombres para las relaciones que tengan con el resto de módulos. Si le pasa un nombre erróneo al servidor este devuelve un error indicando que no existe para ese módulo una relación con ese nombre.

El valor devuelto por la función "*set_relationship*" es el tipo complejo "*new_set_relationship_list_result*", cuya definición es:

```
- <xsd:complexType name="new_set_relationship_list_result">
  - <xsd:all>
      <xsd:element name="created" type="xsd:int"/>
      <xsd:element name="failed" type="xsd:int"/>
      <xsd:element name="deleted" type="xsd:int"/>
    </xsd:all>
  </xsd:complexType>
```

Donde:

- *created*: Número de relaciones que fueron creadas.
- *failed*: Número de relaciones que fallaron al intentar crearse.
- *deleted*: Número de relaciones fueron borradas.

Como se ha comentado previamente, existen tres tipos de relaciones: uno-a-uno, uno-a-muchos y muchos-a-muchos. El primer tipo de relación se establece desde la ventana de edición de una cuenta, donde se muestra el nombre del registro con el que está relacionado. Para los otros dos tipos de relaciones, se crea una nueva ventana donde se muestra una lista con los registros de otro módulo (como por ejemplo Contactos) asociados a la cuenta. En la Figura 29 se muestra el diagrama de secuencia para realizar una relación entre una cuenta y otro módulo:

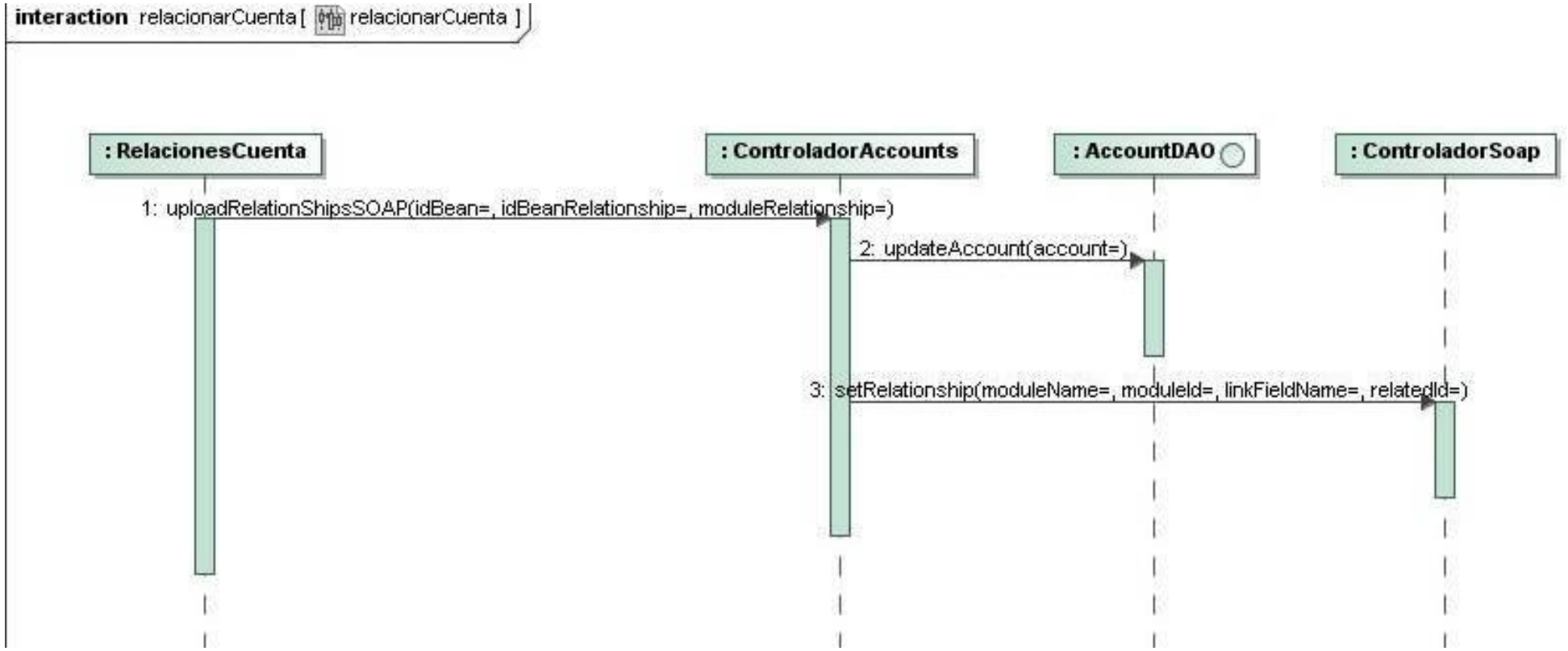

Figura 29. Diagrama de secuencia relacionar cuenta

Para terminar se implementa la lógica necesaria para descargar del servidor las relaciones que ya existan para una cuenta. La función de los servicios web de SugarCRM que nos permite realizar esta acción es "*get_relationships*", definida en la Figura 30:

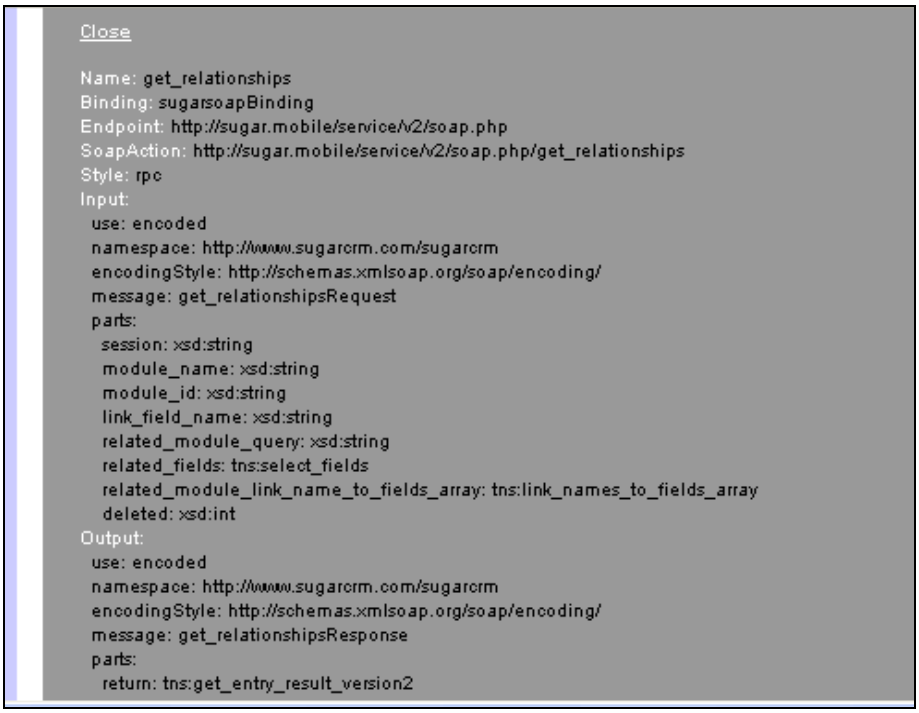

Figura 30. Definición de la función "get_relationships"

Los parámetros de esta función son:

- *session*: Identificador de sesión obtenida al realizar el login.
- *module_name*: Nombre del módulo del registro que va a realizar la relación.
- *module_id*: El identificador del registro del módulo especificado en module_name.
- *link_field_name*: Nombre del tipo de relación que se desea realizar.
- *related_module_query*: Consulta que puede ser añadida para filtrar el resultado devuelto de los registros que se encuentran relacionados.
- *related_fields*: Lista de los campos que deseamos obtener de los registros relacionados.
- *related_module_link_name_to_fields_array*: Lista de nombre de relaciones para obtener información de otras registros relacionados con los registros descargados. Es opcional.

- *delete*: Si el valor es 0 los registros borrados no son incluidos, si es 1 los registros borrados son incluidos.

El valor devuelto por esta función es un tipo complejo llamado "*get_entry_result_version2*" definido como:

```
- <xsd:complexType name="get_entry_result_version2">
  - <xsd:all>
      <xsd:element name="entry_list" type="tns:entry_list"/>
      <xsd:element name="relationship_list" type="tns:link_lists"/>
    </xsd:all>
  </xsd:complexType>
```

Donde:

- *entry_list*: Corresponde a la lista de los registros recibidos.

- *relationship_list*: La información de los registros relacionados con los registros descargados que fueron indicados en el parámetro "*related_module_link_name_to_fields_array*".

En la Figura 31 se muestra el diagrama de secuencia para realizar la descarga de registros relacionados:

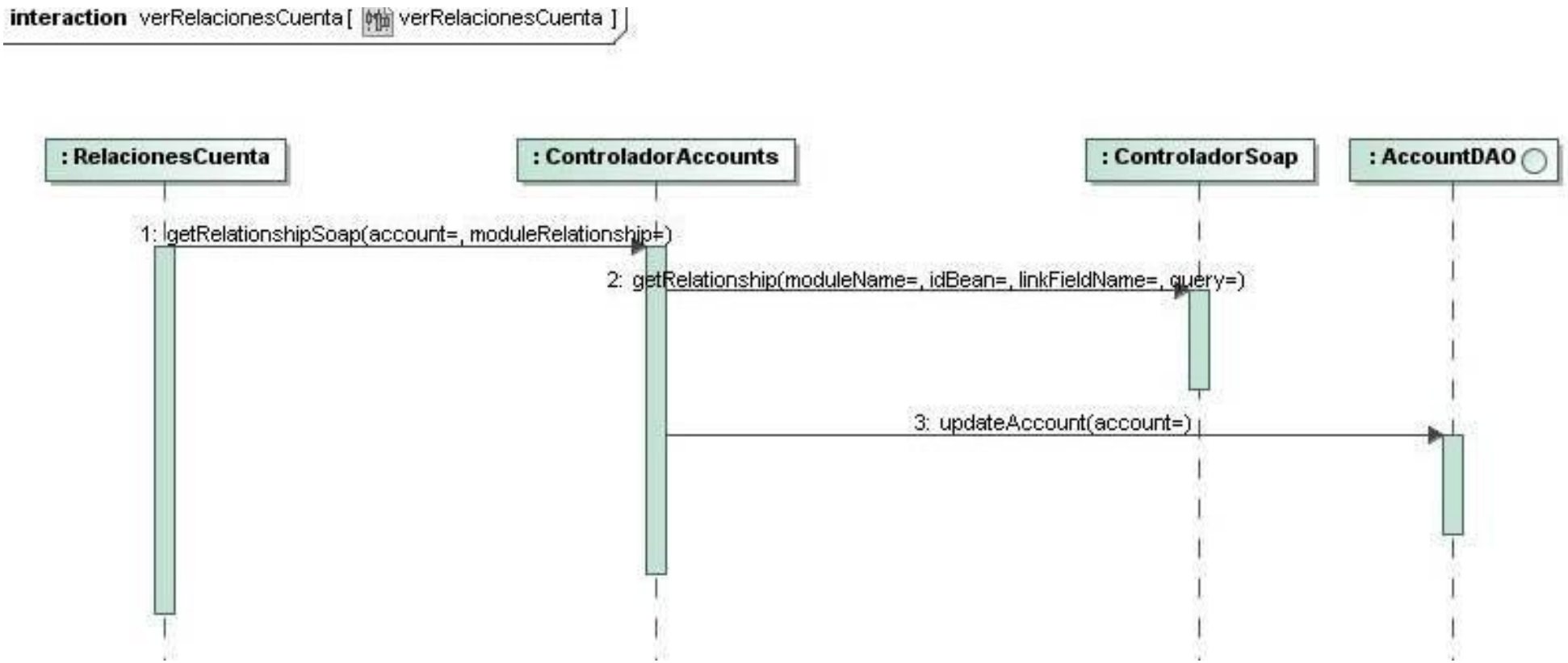

Figura 31. Diagrama de secuencia ver relaciones de una cuenta

Recordar que en la iteración anterior se mostró el método "*actualizaRelationship*" de la clase ControladorAccounts. Este método se encarga de actualizar el valor del identificador de una cuenta que esté relacionado con cualquier otro registro de otro módulo. Esto se debe a que cuando creamos una nueva cuenta, que se inicializa con un identificador temporal, es posible relacionarla con otros registros antes de que obtengamos del servidor el identificador final que le ha asignado. Por lo tanto, una vez que se obtiene la respuesta del servidor con el identificador final se debe informar al resto de registros con los que esa cuenta está relacionada de que su identificador ha cambiado, para que ellos también lo actualicen en sus listas. Si no se hace esto, se daría el caso de que las relaciones que se tenían entre esa cuenta y sus registros relacionados se perderían, ya que estos últimos tendrían el identificador temporal para referenciar a la cuenta, en vez del identificador real asignado por el servidor.

También se comentó en la 3ª iteración de la fase de elaboración que existe una característica especial cuando se muestra la vista de detalles de una cuenta. Esto se debe a que en una cuenta lo que se almacena son los identificadores de los registros a los que está enlazados, y no la información de estos en sí (nombre, usuario asignado, etc.), por lo que cabe la posibilidad de que cuando se intente acceder a esta información, el registro en concreto no se encuentre descargado en el dispositivo y no sea posible acceder a él. La aplicación está diseñada para que cuando suceda este caso automáticamente realice una llamada al servidor para descargarse el registro (o los registros) con los que se encuentra relacionados una cuenta. La función usada es "*get_entries*" definida en la Figura 32:

Figura 32. Definición de la función "get_entries"

Los parámetros de entrada son:

- *session*: Identificador de sesión obtenida al realizar el login.
- *module_name*: Nombre del módulo del registro que va a realizar la relación.
- ids: Lista de identificadores a descargar.
- *select_fields*: Lista de campos a incluir en el resultado.
- *link_name_to_fields_array*: Lista de nombre de relaciones para obtener información de otras registros relacionados con los registros descargados. Es opcional.

El valor devuelto por esta función es un tipo complejo llamado "*get_entry_result_version2*", el mismo que el devuelto por la función "*get_relationships*".

Al finalizar esta iteración, ya tenemos la funcionalidad necesaria implementada en nuestra aplicación para establecer relaciones entre una cuenta y el resto de módulos con los que esté relacionado, así como la descarga de registros relacionados con una cuenta. En la Figura 33 se muestra la ventana correspondiente a las relaciones que tiene una cuenta con el módulo Contactos:

Figura 33. Ventana de los contactos relacionados con una cuenta

Para la siguiente iteración se realizará la implementación necesaria para borrar una cuenta desde el dispositivo móvil.

3.3.3 3ª Iteración (Duración – 2 días):

En esta iteración se realizará la implementación de la eliminación de una cuenta, que se realiza en dos pasos: eliminación de la cuenta en el dispositivo y eliminación de la cuenta en el servidor.

Para eliminar la cuenta del servidor se utiliza la misma función usada para crear y actualizar una cuenta, "set_entry", pero en esta ocasión se le añade al parámetro "*name_value_list*" el campo "*delete*d" con valor "*true*", indicando al servidor que debe borrar ese registro. La Figura 34 muestra el diagrama de secuencia para la eliminación de una cuenta:

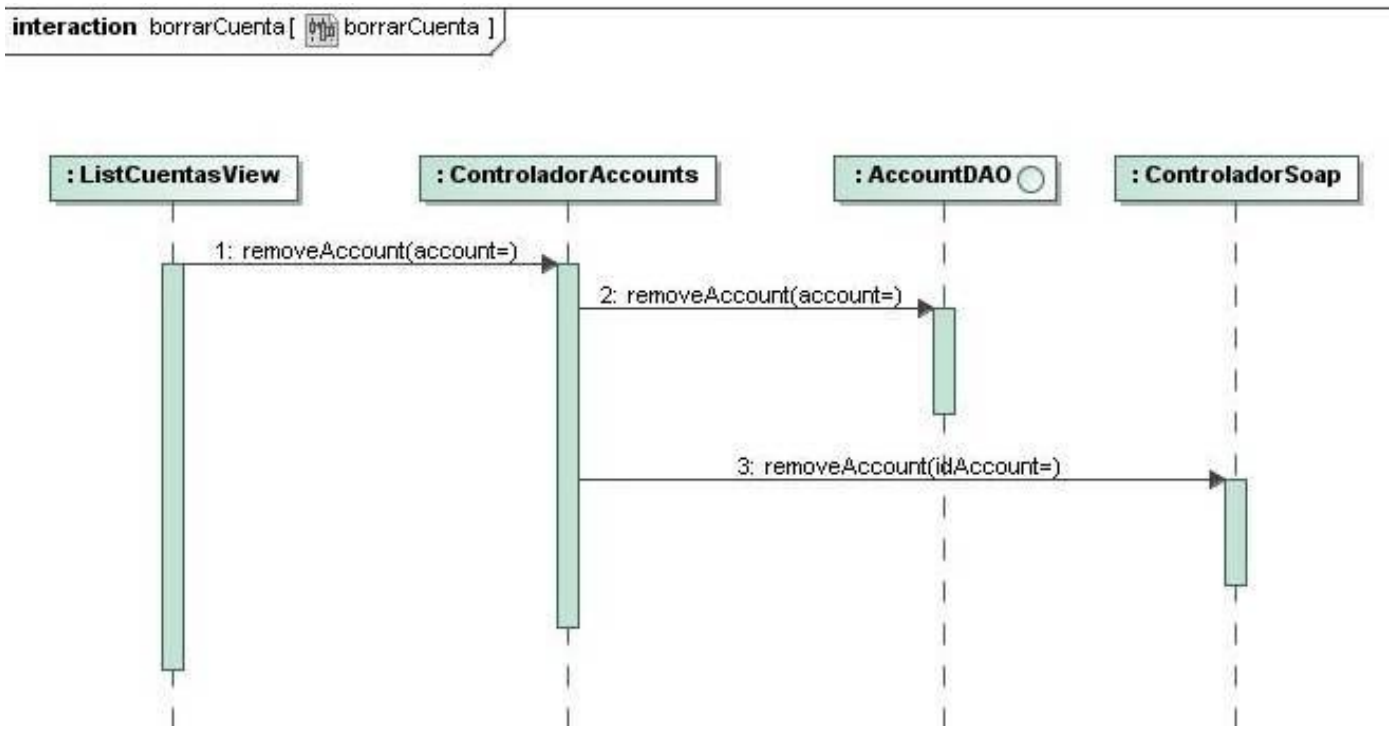

Figura 34. Diagrama de secuencia borrar cuenta

3.3.4 4ª Iteración (Duración – 2 meses):

En este momento se encuentra implementada toda la funcionalidad referente al módulo Cuenta. Debido a que el resto de módulos se implementan de una forma parecida (prácticamente lo que varía son los campos existentes para cada registro y las relaciones con otros módulos) en esta iteración no se va a comentar los pasos seguidos para implementarla, ya que incluso las funciones usadas para interactuar con el servidor son las mismas, cambiando lógicamente los parámetros enviados al servidor. Lo que si merece la pena comentar es que debido a la semejanza en la implementación entre los distintos módulos de la aplicación (Cuentas, Contactos, etc.) se aplica el patrón Template Method para la creación de los controladores. De esta forma, se crea la clase genérica "*Controlador*" donde se implementa todo el código genérico para los controladores, y se crean unos métodos abstractos implementados por las clases que hereden de Controlardor (ControladorAccounts, ControladorContacts, etc.) dónde se realizan las funciones específicas para cada módulo. También se aplica el patrón Modelo-Vista-Controlador separando la implementación de las interfaces gráficas de la lógica de negocio.

En la siguiente iteración se explica una parte muy importante de la aplicación, y es la referente al funcionamiento de la misma en modo offline.

3.3.5 5ª Iteración (Duración – 4 días):

Uno de los objetivos que se enumera en la sección 2.1 es la posibilidad que ofrece la aplicación de poder seguir trabajando con ella a pesar de que el dispositivo no tenga cobertura suficiente para comunicarse con el servidor. En esta iteración se plantea la solución implementada para que el usuario de la aplicación pueda seguir usándola sin que perciba que el dispositivo se ha quedado sin cobertura.

Tal y como se explicó en la fase de elaboración, un usuario puede descargar registros de distintos módulos desde el servidor y luego acceder a ellos sin que se tenga que volver a producir otra llamada al servidor. Para ello, se explica en esa fase como la aplicación descarga los distintos registros y luego los almacena en memoria persistente. Esto permite que, por un lado, las llamadas al servidor sean las mínimas necesarias, disminuyendo por tanto el uso de la radio del dispositivo e incrementando así la duración de la batería; por otro lado, permite seguir trabajando con los registros que tenga descargados a pesar de que el dispositivo se quedara sin batería. Pero aquí surge un pequeño problema, y es que si bien la visualización de los listados de registros de los distintos módulos, así como la visualización de los detalles de un registro, no necesitan realizar

consulta alguna al servidor, si se realiza un operación de modificación de algún registro o de creación de un nuevo registro, ésta si debe ser notificada al servidor, porque si no la información disponible en el dispositivo no sería coherente con la que hay en el servidor. Por lo tanto hay que plantear una solución para que la aplicación pueda enviar las modificaciones realizadas al servidor en el momento que vuelva a tener cobertura el dispositivo.

La solución a este problema está basada en el concepto de "*bitácora*" usada en los gestores de bases de datos. La idea es que cuando se realiza cualquier operación tanto de creación/modificación/eliminación de registros como de creación/modificación/eliminación de relaciones entre registros, éstas antes de ser enviadas al servidor se guardan en una bitácora que es almacenada en memoria persistente. La bitácora es implementada como una lista FIFO, asegurando que las operaciones se envían al servidor en el mismo orden que se realizan. Cada entrada de la lista contendrá la acción a realizar, y no será borrada hasta que la operación sea confirmada por el servidor, ya sea confirmando su realización ó avisándonos del error por el que no pudo realizar la operación. De esta manera, si el dispositivo se queda sin cobertura, las operaciones realizadas por el usuario son reflejadas inmediatamente en la aplicación (significando esto que si el usuario asigna un nuevo contacto a una cuenta, al mirar el listado de usuarios relacionados a esa cuenta aparecerá el último asignado, aunque esa información no haya sido todavía enviada al servidor) y se almacenará en la bitácora para que, cuando el dispositivo recupere la cobertura, la aplicación pueda enviar esas modificaciones al servidor.

Otro aspecto a tener en cuenta es el asunto de los identificadores de registros temporales creados por la aplicación. Como ya se comentó en la segunda iteración de esta fase, cuando se crea un nuevo registro se le asigna un identificador temporal hasta que el servidor devuelva el identificador real que le ha asignado. Esto implica que si una de las operaciones almacenadas en la bitácora corresponde a crear un nuevo registro de un módulo, cuando el servidor responda a esa petición devolviendo el identificador del registro creado, además de realizar las operaciones de actualización que ya se explicaron en la segunda iteración, se actualiza cualquier otra operación de la bitácora que afecte al registro recién creado, ya que el identificador que está almacenado en la bitácora corresponde con el temporal creado por el dispositivo, y no el devuelto por el servidor, y si se realiza una petición al servidor sin modificar ese identificador temporal por el real, el servidor devolverá un error indicando que no existe en su base de datos un registro con ese identificador temporal asignado por la aplicación.

Para terminar, todo lo explicado hasta ahora surge por el hecho de que el dispositivo en cualquier momento puede encontrarse sin cobertura, pero, ¿cómo puede enterarse la aplicación de que el dispositivo no tiene

cobertura? Para ello, la API de BlackBerry incorpora dos interfaces: "*WLANConnectionListener*" y "*RadioStatusListener*".

La primera interfaz tiene dos métodos llamados "*networkConnected()*" y "*networkDisconnected(int reason)*". El primer método es invocado cuando el dispositivo ha sido conectado a una red WLAN; el segundo método es invocado cuando el dispositivo es desconectado de una red WLAN o cuando un intento de conexión a una red WLAN ha fallado, indicando el parámetro "*reason*" el motivo de la desconexión.

La segunda interfaz tiene varios métodos, aunque la aplicación implementa los siguientes:

- "*void networkServiceChange(int networkId, int service)*", invocado cuando los servicios ofrecidos por la red cambian.
- **"***void networkStarted(int networkId, int service)*", invocado cuando la radio es encendida o cuando se produce un cambio a una nueva red.
- "*void networkStateChange(int state)*", invocado cuando se produce un cambio en el estado de la red.
- "void radioTurnedOff()", invocado cuando la radio es apagada.
- "void signalLevel(int level)", invocado cuando la intensidad de la señal recibida cambia.

La clase "*ControladorCobertura*" implementa estas dos interfaces, y es la encargada de avisar al resto de componentes de la aplicación cuándo se produce un cambio en la cobertura del dispositivo. Para ello la clase se registra como oyente en el sistema ejecutando las siguientes instrucciones:

```
WLANInfo.addListener(controladorCobertura);
UiApplication.getUiApplication().addRadioListener(controladorCobertura);
```

Donde "*WLANInfo*" contiene la información referente a WLAN e invocará los métodos de la interfaz "*WLANConnectionListener"* y el método "*addRadioListener*" invocará los métodos de la interfaz "*RadioStatusListener*".

Aquí finaliza esta iteración. En la siguiente iteración se realiza la implementación necesaria para actualizar las modificaciones realizadas en el servidor sobre registros que estén descargados en el dispositivo.

3.3.6 6ª Iteración (Duración – 1 semana):

En esta iteración se realiza la implementación necesaria para que la aplicación pueda actualizar los valores de los registros que tenga descargados y que hayan sido modificados en el servidor por otra aplicación.

Para empezar, se sabe que todos los registros de cualquier módulo en SugarCRM contienen un campo donde se muestra la hora en la que se realizó la última modificación, así como otro campo con el identificador de usuario que realizó esa modificación. Por lo tanto, para saber si un registro guardado en el dispositivo ha sido modificado externamente, tan solo hay que realizar una consulta usando la función "*get_entry_list*" explicada previamente, indicándole en el parámetro "*query*" qué registros son los que interesan, la hora mínima desde la que se realizó la última comprobación e indicarle que esa modificación no la debe haber realizado el usuario que inicie sesión en el dispositivo.

El usuario puede obtenerse de la clase "*ControladorConfiguración*", ya que es quien almacena el usuario que actualmente ha iniciado la sesión en la aplicación. La clase encargada de realizar las actualizaciones es "*ControladorActualizaciones*". Esta clase tiene un método llamado "*checkUpdateBeans()*" que se encarga de avisar a todos los controladores de módulos existentes en el dispositivo (Cuentas, Contactos, etc.) para que realicen una llamada al servidor que compruebe si alguno de sus registros ha sido modificado. Esta comprobación es realizada por la aplicación cada vez que se inicia su ejecución y si se cumplen dos condiciones:

- El usuario tiene guardada la sesión. Esto es necesario ya que cuando el usuario realiza un cierre de sesión todos los registros de los distintos módulos almacenados en memoria son borrados, por lo tanto no habría registro alguno sobre el que realizar la actualización. Esta acción se realiza por motivos de seguridad, ya que si no se eliminan otro usuario podría realizar un inicio de sesión y ver información restringida descargada por el usuario anterior.

- El tiempo transcurrido desde la última actualización es como mínimo de 30 minutos. Para ello, la aplicación guardará la hora de la última actualización realizada y será actualizada cada vez que se realice una actualización.

La hora en la que se realiza la actualización no es obtenida del dispositivo, sino que se realiza una llamada al servidor para obtener su hora actual en GMT. La llamada usada es "*get_server_info*" definida en la Figura 35:

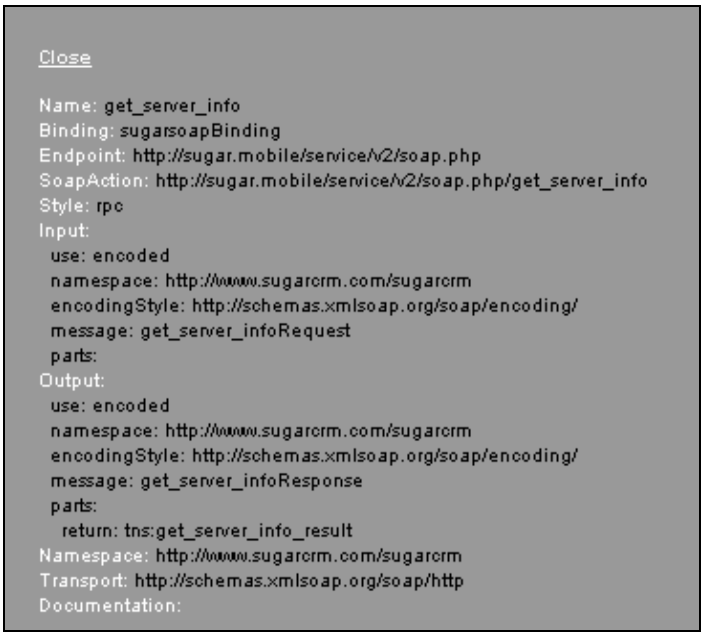

Figura 35. Definición de la función "get_server_info"

Esta función no tiene parámetros de entrada y devuelve un tipo complejo llamado "*get_server_info_result*" definido como:

```
− <xsd:complexType name="get_server_info_result">
  − <xsd:all>
      <xsd:element name="flavor" type="xsd:string"/>
      <xsd:element name="version" type="xsd:string"/>
      <xsd:element name="gmt_time" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
```

Donde:

- "*flavor*": Corresponde con la edición de SugarCRM que se está ejecutando (Community, Professional ó Enterprise).
- "*version*": Corresponde con el número de versión de SugarCRM que se está ejecutando.
- "*gmt_time*": La hora actual del servidor en GMT.

Una vez realizada la consulta al servidor para saber que registros han sido modificados, no se descarga directamente la información, sino que se marca ese registro en concreto como "sucio", de manera que cuando el usuario vea la vista de detalles de ese registro la aplicación comprueba que el registro se encuentra desactualizado y, automáticamente, descargará la información desde el servidor actualizando el registro y mostrándole al usuario la nueva información descargada.

En la siguiente iteración se dota a la aplicación de la funcionalidad necesaria para poder eliminar registros almacenados cuando la memoria disponible del dispositivo llegue a umbrales críticos.

- *7ª Iteración (Duración – 3 días):*

Tal y como se explicó en la Fase de Elaboración, el número de manejadores de referencia persistente es limitado, así como la cantidad de memoria disponible del dispositivo, por lo tanto, dada la naturaleza de la aplicación, donde la descarga y almacenamiento en memoria persistente de registros del servidor es habitual, es de suponer que llegue un momento en que la memoria del dispositivo (ya sea por el número de manejadores o por la cantidad física de memoria disponible) se encuentre en una situación crítica debido a su escasez. Se debe recordar que en el dispositivo existen más aplicaciones ejecutándose y compartiendo (y consumiendo) los mismos recursos que esta aplicación. Para evitar esta situación, existe la interfaz "*LowMemoryListener*", que define un método llamado "*freeStaleObject(int priority)*" que es llamado por la clase "*LowMemoryManager*" cada vez que se produce una situación de memoria libre baja. El parámetro "*priority*" puede tener los siguientes valores:

- *LOW_PRIORITY*: Para este valor, los oyentes sólo deben eliminar objetos que no son críticos para ellos. Por ejemplo, el navegador de BlackBerry elimina las páginas que tenga cacheadas cuando es llamado con esta prioridad.

- *MEDIUM_PRIORITY*: Para este valor, los oyentes deben eliminar, junto con los del nivel anterior, objetos que son muy viejos o antiguos. Por ejemplo, la aplicación de correo de BlackBerry elimina los mensajes antiguos cuando es llamado con esta prioridad.

- *HIGH_PRIORITY*: Para este valor, los oyentes deben eliminar, junto con los del nivel anterior, todos los objetos que no considere indispensables, hasta conseguir reunir la cantidad de memoria necesaria.

La clase "*ControladorMemoria*" implementa la interfaz "*LowMemoryListener*", y se registra como oyente en la clase "*LowMemoryManager"* a través del método "LowMemoryManager.*addLowMemoryListener(LowMemoryListener lml)*". Cuando la clase "*LowMemoryManager*" llama al método "*freeStaleObjet*(Object o)", se realiza una llamada a todos los controladores indicándole cual es el nivel de prioridad y el identificador del usuario actual. Cada controlador realiza a su vez una llamada a su DAO correspondiente pasándole los parámetros recibidos, y el DAO realizará una de las siguientes acciones:

- Si la prioridad es baja, se eliminarán todos los registros que estén desactualizados. Estos registros habrán sido marcados previamente debido a que se han modificado sus datos en el servidor y, como deben descargarse de nuevo, pueden ser borrados del dispositivo, ya que la información que contienen no es coherente con la que tiene el servidor.

- Si la prioridad es media o alta, se eliminarán todos los registros que estén desactualizados y todos aquellos registros que estén asignados a otros usuarios distintos al usuario que inicio sesión en la aplicación.

Para cada objeto liberado, se ejecuta el método "*LowMemoryManager.markAsRecoverable(Object o)*", indicándole al SO la cantidad de memoria que hemos liberado. Se debe tener en cuenta que esta llamada no implica que directamente se libere la memoria, sino que el SO la liberará cuando considere oportuno. Esto se debe a que cuando se lanza el liberador de memoria el rendimiento del resto de las aplicaciones de la BlackBerry suele verse muy deteriorado, debido a que los recursos consumidos por este proceso son altos. Por eso, generalmente el SO realiza esta tarea cuando el dispositivo se encuentra en un estado inactivo o de poco uso como, por ejemplo, cuando detecta que el dispositivo se encuentra dentro de la funda o no está siendo usado por el usuario desde hace algún tiempo.

En este punto termina la fase de construcción. La aplicación ya se encuentra en una etapa totalmente funcional y con toda la funcionalidad deseada implementada, por lo que se puede pasar a la siguiente fase del proceso unificado.

3.4 Fase de Transición

En esta fase se realizan las pruebas betas a la aplicación y el despliegue de la misma, pero para este proyecto no se incluirán estas pruebas, ya que el alcance del mismo abarca hasta la fase de construcción.

4 CONCLUSIONES Y VÍAS FUTURAS

La finalidad de este proyecto era conseguir que un usuario pudiera acceder desde un dispositivo BlackBerry a la información almacenada en SugarCRM desde cualquier lugar en el que se encontrara. Para ello, se ha desarrollado una aplicación que se comunica con SugarCRM a través de los servicios web que ofrece, usando para ello diferentes herramientas como el plug-in de RIM para Eclipse, junto con la versión 4.5 de la API BlackBerry, usado para desarrollar la aplicación BlackBerry, y el uso de mensajes SOAP para comunicar al servidor SugarCRM y al dispositivo, usando para la creación de los mensajes de petición, y recepción de los mensajes de respuesta, la librería KSOAP para J2ME. Además de esta comunicación entre dispositivo y servidor, se le han añadido a la aplicación funcionalidades como la posibilidad de funcionar en modo offline, usando para ello una bitácora donde se va almacenando ordenadamente las operaciones que se van realizando para posteriormente enviarlas al servidor, así como reducir en lo posible tanto el uso de la radio como el consumo de batería y memoria del dispositivo.

Quizás el punto más positivo haya sido la aplicación del Proceso Unificado para el desarrollo de la aplicación. A pesar que durante los años de carrera se ha explicado los beneficios de hacer uso de un proceso de desarrollo para conseguir desarrollar una aplicación correcta, y correctamente, bien es cierto que no se puede hacer uno a la idea de lo cierto que es hasta que se introduce en un proyecto real. En el desarrollo de este proyecto se ha podido comprobar cómo puede llegar a complicarse el desarrollo de una aplicación por diversas causas (la no compresión de los requisitos, la adición de nuevos requisitos a mitad de proyecto, problemas inesperados sobre funcionalidad de la aplicación que se consideraban resueltos, etc.) y cómo el ir solucionando pequeños problemas iteración tras iteración, abarcando primero las situaciones más críticas que podrían haber hecho peligrar al proyecto, y haciendo uso de patrones de diseños para resolver problemas comunes en el desarrollo de aplicaciones, permite llegar al final del desarrollo del proyecto obteniendo la aplicación que se deseaba.

El punto más negativo ha aparecido sobre todo con la implementación de las interfaces gráficas de la aplicación, debido a que no existe ningún editor gráfico para crearlas, por lo que todo se realiza a través de código. Esto suponía que cada vez que se quería comprobar el "look" de la aplicación te obligaba a lanzar el simulador, lo que podía llegar a suponer un consumo de tiempo significante, sobre todo cuanto implementabas un componente que debido a su complejidad te obligaba a lanzar repetidamente el simulador para comprobar su correcta apariencia.

Por último comentar algunas de las mejoras que se le podrían añadir en un futuro a la aplicación, por ejemplo:

- Modificar las acciones de la aplicación según el nivel de batería. Se podría añadir a la aplicación la capacidad de dejar de usar la radio (igual que cuando no tiene cobertura suficiente) cuando el nivel de batería se encuentre por debajo de un mínimo. Para ello tendría que registrarse en el sistema invocando a "*Application.addSystemListener(listener)*" e implementado la interfaz "*SystemListener*", que contiene los métodos invocados cuando se produce un cambio en el nivel de batería.

- Añadir la tecnología push a la aplicación. A pesar de que el uso de la radio y consumo de batería se ha reducido en lo posible, la aplicación para poder actualizarse con los cambios realizados en el servidor por otros usuarios o aplicaciones, necesita realizar periódicamente llamadas al servidor preguntando si se han modificado algún registro de los que tiene descargados actualmente. Esto podría evitarse si se hiciese uso de la tecnología push, usada por ejemplo en la aplicación de mensajes de BlackBerry, que consiste en que el servidor, sin una llamada previa por parte del dispositivo, sea quien avise al dispositivo de los cambios sucedidos, si esos cambios fueran relevante para el dispositivo. En este escenario, la aplicación tan solo tendría que estar escuchando los mensajes que le enviara el servidor para actuar en consecuencia.

- La nueva versión de la API 5.0 BlackBerry incluye SQLite, una librería para bases de datos relacionales. Está diseñada para hacer un uso eficiente de los recursos de memoria, tanto de la memoria interna del dispositivo como de cualquier memoria SD externa. El único problema es que sólo está disponible para dispositivos que ejecuten la versión 5.0 o posterior del SO, por lo que este planteamiento no funcionaría en dispositivos antiguos.

- Mejora de las estructuras de datos usadas por la aplicación. Las estructuras usadas para almacenar los datos han sido tablas hash y vectores ordenados. El problema aparece con el segundo tipo de estructura de datos, ya que si se incremente el número de registros almacenados en ellos el tiempo de ejecución para realizar una consulta o modificación de la estructura crece también linealmente, lo que supondría para el usuario un decremento en el rendimiento. Para evitar esto podrían cambiarse estos vectores ordenados por unas estructuras arbóreas, cuyo tiempo de ejecución tiene un crecimiento logarítmico, resultando mucho más eficiente que los vectores.

- Añadir a la aplicación la funcionalidad de internacionalización. Todos los textos mostrados por la aplicación están en español. Si se quiere conseguir que la aplicación sea usada por la mayor cantidad posible de usuarios, se debería traducir todos esos textos a distintos idiomas. La API BlackBerry provee un conjunto de clases en el paquete "net.rim.device.api.i18n" que nos proveen la funcionalidad necesaria para soportar la internacionalización de manera sencilla en aplicaciones BlackBerry.

5 BIBLIOGRAFÍA

- Anthony Rikz (2009). Beginning BlackBerry Development. Apress
- Chris King (2009). Advanced BlackBerry Development. Apress
- Robert Kao, Dante Sarigumba (2009). BlackBerry for Dummies. Wiley Publishing, 3ª Edición.
- Craig Larman (2003). UML y Patrones. Prentice Hall, 2ª Edición.
- Web oficial desarrolladores BlackBerry .

  Español➔ http://es.blackberry.com/developers/ (Consultado el 1 de Julio de 2010)

  Inglés ➔ http://na.blackberry.com/eng/developers/ (Consultado el 1 de Julio de 2010)
- Web oficial Eclipse. http://www.eclipse.org/ (Consultado el 1 de Julio de 2010)
- Web oficial KSOAP2. http://ksoap2.sourceforge.net/ (Consultado el 1 de Julio de 2010)
- Web oficial SugarCRM. http://www.sugarcrm.com/crm/ (Consultado el 1 de Julio de 2010)
- Web oficial estándar Servicios Web . http://www.w3.org/ (Consultado el 1 de Julio de 2010)

6 ANEXO I – MANUAL DE USUARIO

En este anexo se incluye el manual de usuario para la aplicación desarrollada. La primera vez que se ejecute la aplicación se mostrará la ventana de configuración de la misma. Se introducirán los datos correspondientes a la dirección url donde se encuentra el servidor SugarCRM, así como indicarle a la aplicación cómo debe salir a la red, ya sea por BIS o por WiFi. Una vez se pulse el botón "Guardar" la aplicación realiza una comprobación de la url introducida, como se muestra en la Figura 36:

Figura 36. Ventana de configuración

Si la url no es correcta, la aplicación lo indica mostrando un mensaje de error, en caso contrario muestra información sobre el servidor, como se muestra en la Figura 37:

Figura 37. Datos de la versión del servidor

Una vez configurada la aplicación, se muestra la ventana de login, mostrada en la Figura 38:

Figura 38. Ventana de login

Se da la opción de guardar la sesión de usuario para las siguientes ejecuciones de la aplicación. Si el nombre o password del usuario no son correctos, se muestra un mensaje de error indicándolo. Si los datos son correctos, se muestra la ventana principal de la aplicación. En ella se muestra una lista en la que seleccionar los módulos disponibles de la aplicación, así como la información más relevante de cada módulo para el usuario, como se muestra en la Figura 39:

Figura 39. Ventana principal

Pulsando en el icono de cualquier módulo, se muestra la ventana con el listado de registros descargados para ese módulo. Pulsando la tecla "menú" aparecen las siguientes opciones, como se muestra en la Figura 40:

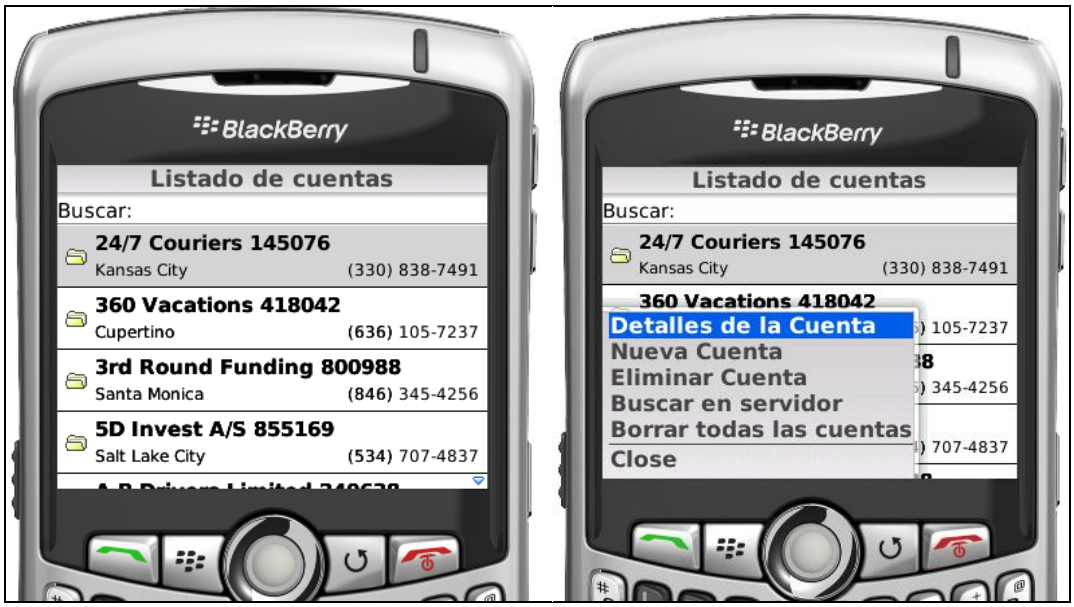

Figura 40. Ventana listado de cuentas

Si selecciona la opción "Buscar en servidor" se muestra una ventana desde la que puede realizar una búsqueda en el servidor de cuentas según el filtrado deseado, como se puede ver en la Figura 41:

Figura 41. Ventana de búsqueda en el servidor

Seleccionando la opción "Eliminar Cuenta" borrará la cuenta tanto en el dispositivo como en el servidor. Si se desea borrar todas las cuentas tan solo del dispositivo seleccione la opción "Borrar todas las cuentas".

La opción "Detalles de cuenta" muestra una nueva ventana con toda la información referente a esa cuenta. Como el usuario asignado corresponde a una relación entre el módulo cuenta y el módulo usuario, si el usuario asignado a esa cuenta no está descargado en el dispositivo, la aplicación lo descarga automáticamente, como se muestra en la Figura 42.

Figura 42. Ventana de detalles

Si se pulsa la tecla menú desde esta ventana, aparecerán en el menú los distintos módulos con los que está relacionada la cuenta, así como la opción de editar la cuenta. Si se pulsa en uno de ellos, aparecerá una ventana con un listado de los registros de ese módulo relacionados con la cuenta. De nuevo, si alguno de esos registros no se encuentran en el dispositivo, la aplicación los descargará automáticamente, como se muestra en la Figura 43:

Figura 43. Ventana de contactos relacionados con una cuenta

Al pulsar la tecla menú desde esta ventana se nos mostrarán las opciones de "Sincronizar relaciones", "Añadir contacto" y "Eliminar Contacto". La primera opción sirve para sincronizarse con la información actual del servidor, de manera que la información existente tanto en el servidor como el dispositivo sea coherente. La segunda opción sirve para relacionar un nuevo contacto con la cuenta, al pulsar en esta opción se muestra la ventana de listado de contactos, dando la opción de seleccionar uno de ellos para realizar la relación. La tercera opción sirve para eliminar la relación establecida entre un contacto y la cuenta. La Figura 44 muestra estas opciones y los contactos relacionados:

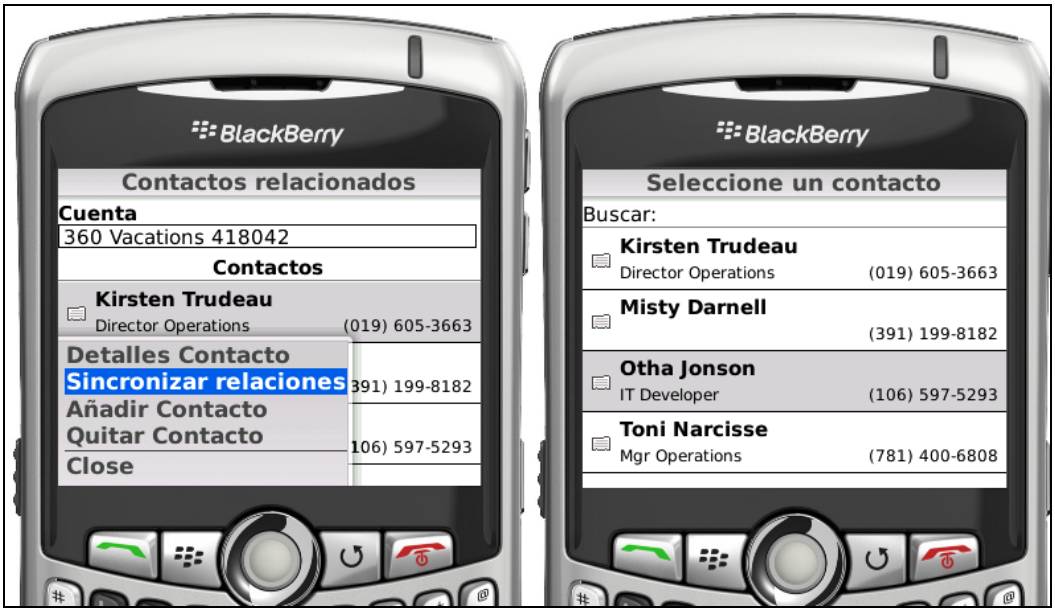

Figura 44. Ventana de contactos relacionados con una cuenta

Desde la vista de detalles de la cuenta, si se pulsa la opción de editar la cuenta, se muestra una ventana parecida a la de detalles de cuenta, pero con los campos editables, como se muestra en la Figura 45:

Figura 45. Ventana de edición de cuenta

Las opciones disponibles para el resto de módulos son semejantes a las comentadas para el módulo cuenta.

Para terminar, si se pulsa la tecla menú desde la ventana principal se muestran las opciones de "Cambiar configuración" y "Cerrar Sesión". La primera opción mostraría de nuevo la ventana de configuración para poder cambiar tanto la url del servidor como la forma en la que sale la aplicación a la red. La segunda opción cerraría la sesión actual del usuario, eliminando todos los registros descargados al dispositivo. La Figura 46 muestra estas opciones:

Figura 46. Menú de la ventana principal

7 ANEXO II – EJEMPLO FICHERO WSDL

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<definitions xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://www.sugarcrm.com/sugarcrm"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://www.sugarcrm.com/sugarcrm">

<types>
      <xsd:schema
       targetNamespace="http://www.sugarcrm.com/sugarcrm">
      <xsd:import
       namespace="http://schemas.xmlsoap.org/soap/encoding/" />
      <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/" />
      <xsd:complexType name="note_attachment">
            <xsd:all>
                        <xsd:element name="id" type="xsd:string" />
                        <xsd:element name="filename" type="xsd:string" />
                        <xsd:element name="file" type="xsd:string" />
            </xsd:all>
            </xsd:complexType>

            ……

</types>

http://sugardev.neosistec.com/soap.php/wdsl?wsdl - #<message name="loginRequest">
      <part name="user_auth" type="tns:user_auth" />
<part name="application_name" type="xsd:string" />
</message>
<message name="loginResponse">
            <part name="return" type="tns:set_entry_result" />
</message>

……

  <portType name="sugarsoapPortType">
<operation name="is_user_admin">
  <input message="tns:is_user_adminRequest" />
  <output message="tns:is_user_adminResponse" />
  </operation>
<operation name="login">
  <input message="tns:loginRequest" />
  <output message="tns:loginResponse" />
  </operation>
……..

</portType>
```

```
<binding name="sugarsoapBinding" type="tns:sugarsoapPortType">

        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

        …….

      <operation name="login">
            <soap:operation
            soapAction=http://sugardev.neosistec.com/soap.php/wdsl/lo gin
            style="rpc" />

                <input>
                        <soap:body use="encoded"
                         namespace=http://www.sugarcrm.com/sugarcrm
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
                </input>

                <output>
                        <soap:body use="encoded"
                          namespace="http://www.sugarcrm.com/sugarcrm"
                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
                </output>
      </operation>

      ……..

</binding>

<service name="sugarsoap">
        <port name="sugarsoapPort" binding="tns:sugarsoapBinding">
                <soap:address location="http://sugardev.neosistec.com/soap.php" />
        </port>
</service>

</definitions>
```